2018

Mapreduce and Heterogeneity: Power-Aware Bag-of-Tasks, Framework Parameter Sensitivity, and Dynamic Cluster Aware Framework Configuration

Jessica L. Hartog
Binghamton University--SUNY, jessica.hartog@binghamton.edu

MAPREDUCE AND HETEROGENEITY: POWER-AWARE BAG-OF-TASKS, FRAMEWORK PARAMETER SENSITIVITY, AND DYNAMIC CLUSTER AWARE FRAMEWORK CONFIGURATION

BY

JESSICA HARTOG

BS, State University of New York at Binghamton, 2009
MS, State University of New York at Binghamton, 2011

DISSERTATION

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2018

Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2018

May 22, 2018

Madhusudhan Govindaraju, Chair and Faculty Advisor
Department of Computer Science, State University of New York at Binghamton

Michael J. Lewis, Member
Department of Computer Science, State University of New York at Binghamton

Dmitry V. Ponomarev, Member
Department of Computer Science, State University of New York at Binghamton

Hiroki Sayama, Outside Examiner
Department of Systems Science and Industrial Engineering, State University of New York at Binghamton

Abstract

This dissertation presents the techniques for adaptation of MapReduce frameworks to incorporate heterogeneity-aware scheduling algorithms, an inspection of cluster configurations and how they impact these scheduling algorithms, an analysis regarding how the cluster configuration and the heterogeneity-aware scheduling can work together to minimize turnaround time and/or power consumption of the cluster when executing MapReduce applications, and how these lessons can be applied more broadly to Big Data infrastructure outside of MapReduce that supports multiple Big Data frameworks simultaneously.

Heterogeneity exists in various capacities in any given cluster, from static (Physical and Platform) heterogeneity to dynamic heterogeneity (Transient Data, Transient Applications, and Irregular Hardware Behavior). Within a cluster there are historically several types of mitigation strategies for each of these types of heterogeneity, and each has their pros and cons. We discuss these mitigation strategies and the types of heterogeneity each of these strategies is able to address, as well as the history of related work in the field.

After this, we consider taking host-level metrics and using them to schedule tasks in real time, with a desire to address cluster-wide energy usage. To do this, we consider estimators for power consumption that are available on-chip, namely

temperature. We establish a correlation between CPU temperature and power consumption, then derive a scheduling algorithm that eliminates nodes that are consuming too much power from the pool of schedulable resources. In order to do this we focus on the ability of MapReduce frameworks, constructed as we have constructed the frameworks described in this thesis, to delay binding of tasks to specific workers. We analyze the impacts this has on turnaround time of a MapReduce application, with analysis around setting this threshold properly to reduce impact on turnaround time while shifting power consumption around in the cluster, away from nodes that are over-consuming.

We also address concerns with respect to upgrading a cluster in stages, introducing more physical heterogeneity at various levels, and the types of adjustments that need to be made to MapReduce configurations in order to combat the increased heterogeneity. In particular, we look at the concerns for MapReduce platform mis-configuration and its impacts on turnaround time, analyzing the ways in which these types of errors can be mitigated between incremental platform upgrades. In an effort to address this type of systematic cluster upgrades and the problems with configuration it creates, we introduce a Dynamic Heterogeneity Awareness (DHA) module to our MapReduce framework. This module allows better spreading of tasks by the framework, in order to further improve turnaround time and resource utilization.

Finally we consider the implications for framework and application co-tenancy, and we describe the state of art in these areas. We focus on describing what co-tenancy is, why it's important, and how the state of the art can be expanded to in order to leverage findings from this thesis to make these co-tenant clusters increase

application and framework performance as well as improving these clusters with considerations for energy efficiency.

Dedication

In loving memory of Patricia Ann Leahy, my grandmother and friend who passed away during the course of my studies.

Acknowledgments

I'd like to take a moment to acknowledge my friends and family who have helped me through this dissertation and all of life's challenges that came during my tenure as a PhD student. In particular I'd like to thank my parents (Jan and Peggy Hartog), grandparents (Elsie Hartog, and Patricia and Raymond Leahy), sisters (Heather, Melissa, and Katie Hartog), lab cohorts (especially Renan DelValle), extended family and close friends who have provided a supportive shoulder, a listening ear, and a reminder of the motivations needed to finish this work. I'd also like to thank my advisor (Madhusudhan Govindaraju) for all the challenges he helped me tackle and for the opportunities to demonstrate excellence and work toward the cutting edge during my course of study.

Table Of Contents

List of Tables

List of Figures

Chapter 1

The Problem of Heterogeneity

Heterogeneous computer clusters have always been a challenge since the conception of distributed computing. Unfortunately this complicates the behavior of the systems that run on top of these clusters, creating the need for additional considerations within designing scheduling algorithms for these distributed systems. To fully explore this space, we need to consider some of the different things that contribute to heterogeneity and how they can impact distributed systems.

1.1 Categories of Heterogeneity

There are several types of static heterogeneity in a distributed computing context:

- **Physical** - This type of heterogeneity includes things like hardware configurations, power draw specifications, placement of machines in their racks, wear on storage mediums, and any number of other such differences. Some of the more subtle physical difference in machines can be something like the amount of dust on a fan, imperfections in the thermal conductivity of a heat

sync, or the amount of thermal paste between the processor and the heat sink. Despite best efforts, all distributed systems consist of an array of physically heterogeneous machines. For more on this see Chapter 1.2.

- **Platform** - From the OS and software packages to framework and application versions, any number of differences can exist in the program stacks for Big Data applications. Each of these components likely has varying degrees of difference across the entire cluster. While it is often true that hosts are designed to have exactly the same software across nodes, when managing tens, hundreds, or thousands of machines it becomes easier and easier for machines to fall through the cracks, so to speak, when performing upgrades to different parts of the platform. Additionally, many software updates are not performed simultaneously on all machines, meaning that at any given point in time the software in a distributed system has the potential to have at least one node out of sync.

These static types of heterogeneity can more readily be addressed by schedulers that can take some of these elements into account by host profiling or keeping of historical scheduling information and performance. In recent work, some of this type of accounting is done via machine learning. This however overlooks the dynamic heterogeneity that also exists in distributed systems, and can often be ignored as it is not encountered when profiling of nodes, or may be overlooked as an outlier by machine learning algorithms. Examples of such dynamic heterogeneity include:

- **Transient Data** - Naturally-occurring data sets have varying degrees of heterogeneity distributed throughout them. Uniform data is difficult to come

by in a natural setting, and sometimes even small variances can lead to edge cases in processing that cause stragglers, or in the case of GPGPU programming thread divergence [82].

- **Transient Applications** - A commonality between all big data frameworks is that they run on top of a standard Operating System (usually Linux). There are software updates, processes designed for data backup, metrics reporting applications, and any number of other such applications that may start and stop at different times on different machines in a cluster. We consider these applications to be transient in nature and somewhat unpredictable within a margin of error. These jobs are often scheduled with some fudge factor in order to not generate a sudden flood of data or cause a spike in work when multiple instances of such jobs are scheduled at the same time across the cluster.

- **Irregular Hardware Behavior** - Hardware deteriorates over time, and this deterioration occasionally impacts the performance of hardware. For example, SSDs have a limited number of writes per block, and if a block is no longer able to be written to a consequence is that the hardware (while not failed) needs to perform additional work in order to perform a write of data. This type of behavior is generally unpredictable and can dramatically reduce performance of an application.

Since there are many different forms of heterogeneity, there have been many mitigation strategies invented to ease the burden of heterogeneity in distributed systems.

1.1.1 Mitigation Strategies

The idea that heterogeneity can occur and needs to be mitigated has been explored in a fair amount of related work. Outlined below are several of the proposed solutions for dealing with heterogeneity and which types of heterogeneity they mitigate.

Profiling and Data Skew

Profiling worker nodes has been used to mitigate heterogeneity for a number of big data platforms. By profiling worker nodes, frameworks are able to predict which nodes will be capable of processing more work. Profiling is often done by running either benchmarks or micro-benchmarks in order to determine the capabilities of nodes in the cluster. Once the nodes have been profiled, the amount of work each node is responsible for processing is decided based on the results of the profiling. This is a means of making it such that more capable nodes are expected to process more work than less capable nodes.

Profiling and data placement is effectively able to address physical and platform heterogeneity. However, it is unable to address transient data, transient applications, and irregular hardware behavior. This is because profiling is based on static information related to when the node was profiled, information that is not updated when circumstances change. In this way, it is possible that over time this method of mitigation does not bode well for platform heterogeneity, since platform variations are more likely to occur as time goes on. Similarly, in the event that any of the dynamic heterogeneity elements spring up, a node that was deemed more capable of processing will always be treated that way, and completing more work

despite now being less able to perform the tasks.

Speculative Execution

Another strategy for mitigating heterogeneity is by performing speculative execution. This method is triggered when some tasks in a distributed cluster finish, and some remain running. These tasks are called *stragglers* and they occur frequently within frameworks that rely on batch processing. A typical scenario where speculative execution can be leveraged is where all nodes in a cluster are provided with some work to do, and as some workers finish their work they pick up the same tasks as other workers in hopes that they'll finish first. Whichever worker finishes first sends a message to a coordinator which then terminates the duplicate work. This is able to mitigate physical and platform heterogeneity, as well as transient applications, and irregular hardware behavior. However, this process is less able to handle transient data, as the difficult to process data (that which spurs the straggler) continues to be difficult to process when it is speculatively executed. Namely, this data, if it is similarly sized to other, easier to process, pieces of data, will likely have its processing completed by the first iteration of the task, instead of the speculative task.

Conclusions

The mitigation strategies often employed for handling of static and dynamic heterogeneity are able to handle a fair number of the types of heterogeneity that happen in a large scale distributed system. Notably, when combined, profiling and data skew, in conjunction with speculative execution, can create more problems

than it solves. This is because skewing the data makes the tasks uneven, and if the skew is incorrect because of some dynamic heterogeneity, then the speculative execution has to overcome a much larger hurdle when the straggling task is associated with a larger data set. Therefore, there is no panacea, and there can be naturally occurring types of heterogeneity that can appear in the system that render the currently employed mitigation strategy less effective.

1.2 Heterogeneity and MapReduce

As the MapReduce community has grown, the number and types of applications that are able to run on MapReduce infrastructure has also grown. Examples of such work include sequencing the human genome [49, 36], identifying and cataloging celestial bodies [69], and even for artistic endeavors like ray tracing [58]. As a result of the various and sundry applications to which the MapReduce paradigm has been applied, it stands to reason that the programming model will not be going away any time soon. As a result, dedicated energy efficient MapReduce implementations, especially with respect to heterogeneous clusters are of ever growing importance. Clusters that are being used to harness the power of MapReduce include ad-hoc clusters such as Condor [18] and opportunistic clusters such as MOON [45]. Even traditional clusters such as those that were available in FutureGrid [1] and its successor XSEDE [2], or those that are equipped with specialized equipment, like GPUs as described by He et al. [34] can benefit from scheduling improvements.

In GreenHDFS [38] the MapReduce cluster is separated into Hot and Cold zones. Nodes in the Hot zone are frequently accessed because they host popular data and consist of high power, high performance CPUs. Nodes in the Cold zone are infrequently accessed as they host unpopular data and are energy-conserving nodes. GreenHDFS uses the underutilization of nodes in the cluster to increase utilization in the Hot zone and aggressively shutdown components to combat idleness in the Cold zone, thus producing energy savings. Leverich and Kozyrakis [43] approach conserving power in Hadoop Clusters by utilizing Hadoop's replication strategy to produce a Covering Subset (CS) of the cluster that contains at least one replica of each data-block. This allows nodes not in the CS to be disabled

to conserve power.

Lang and Patel [42] re-interpret this same problem, but instead of leaving the cluster online at all times, with some nodes sleeping, they consider what would happen if the cluster was asleep until a job was queued. Both Leverich and Kozyrakis, and Lang and Patel, discover considerable power savings. The drawback of the approaches set forth in these works is the tight coupling of the file system and the MapReduce framework. This does not allow for utilization of forthcoming green distributed file systems and results in overhead that decreases efficiency in terms of both power and turnaround time. For further discussion of this refer to this lab's previous work MARIANE[27].

Chen et al. [16] test HDFS and how replication, block size, and file size impact energy efficiency. They conclude that where reliable storage systems, separate from HDFS, are deployed alongside Hadoop, replication should be set to 1, as the replication and shuffling mechanisms utilized by HDFS unnecessarily consume power in this case. In this chapter, we require a reliable storage system to use our framework, thus allowing for power conservation beyond what Hadoop may achieve as it wastes power managing various aspects of HDFS.

Wirtz and Ge [75] analyze the use of Dynamic Voltage and Frequency Scaling (DVFS) in a homogeneous cluster in order to improve energy efficiency. They claim that a power-aware cluster is defined by the number of compute nodes and the number of processing cores per node, together with the frequency of the processor cores. We find that this simplification of the cluster fails to take into account heterogeneity and elasticity. Assuming homogeneity also precludes the possibility of different machines being added to an already existing cluster at a later date, the

creation of an ad hoc MapReduce cluster, and reduces the efficiency of a shared cluster. Additionally, allowing multiple jobs to execute on the same machine can cause contention for shared resources and create slow nodes.

Related to this, Chen et al. [16] discovered that slow nodes need to either be removed or assigned less work in order to reduce power consumption when the speedup provided by adding the node is not enough to offset the power penalty for adding the node.

Throughout much of this work it is important to consider the work distribution of various frameworks which addresses both of these problems by splitting the job into m tasks, where m $<<$ the number of worker nodes. Each worker node is then assigned one task. Faster nodes request additional tasks once their initial task is completed. If new machines are added mid-execution, they can request some of the remaining tasks from the queue.

Energy-Proportional Computing as described by Barroso and Hölzle [11] states that we should consider induced energy difference since idle power dominates total power consumption, and was used as a metric for determining energy efficiency in papers throughout the literature [42, 75, 16]. Chen et al. [16] also suggest that in analyzing an energy efficient MapReduce implementation multiple metrics should be used, including, but not limited to: finishing time, energy, and power. We will use these same metrics for reporting results of our experiments.

Wirtz and Ge [75] collected data for Matrix Multiplication, CloudBurst, and Sort to determine the energy efficiency of their framework; Chen et al. [16] also used Sort for this purpose. These same benchmarks were also considered by Zaharia et al. in their work [26]. This set of benchmarks serve to show that there

are different types of MapReduce workloads, and so any decision regarding the efficacy of scheduling on a framework must work for various types of workloads.

At the outset of this work, MARLA [28] had come out of the Binghamton Grid and Cloud computing research lab and had shown that increasing the number of tasks allows for more opportunities to schedule tasks in heterogeneous and load imbalanced clusters. MARLA relies on the following modules:

- *Splitter* - Performs input splitting, ingesting the file and splitting it into a framework-configured number of pieces, where each piece represents a task.
- *TaskController* - Distributes the map and reduce code to the worker nodes by leveraging the underlying shared filesystem, and distributes unprocessed tasks from the task bag, as well as moving completed tasks to the completed task bag.
- *FaultTracker* - Monitors tasks and resubmits them to the task bag if they failed, and manages strikes on workers with failed taks.

Given that a bag of tasks approach to MapReduce increases the opportunities in which work can be scheduled across nodes in a cluster, there are additional open questions that need answering:

- How can we leverage the new-found scheduling opportunities to try and mitigate cluster power consumption?
- How important are the configuration parameters within a MapReduce framework of this type?
- How are platform upgrades impacted by these configuration parameters?

- Can we improve MapReduce scheduling mechanisms in an effort to reduce the dependence on framework parameters when distributing tasks, and still achieve improved application performance?

This dissertation serves as a means of uncovering the answers to these questions.

1.2.1 Thesis Statement

Heterogeneity complicates scheduling in the Big Data landscape, and this dissertation posits and demonstrates that incorporating heterogeneity-aware scheduling algorithms into MapReduce frameworks can provide a mechanism by which turnaround time and/or energy efficiency of the cluster can be improved when running such applications. Along the way we discover that the degree of performance heterogeneity in a cluster influences the turnaround time of MapReduce applications, and this should be taken into consideration when defining the number of tasks in relation to the number of worker nodes. We also explore what happens when defining too many tasks, as they introduce overhead that cannot be ignored at scale. Following this, we build mechanisms that serve to mitigate error prone manual configuration of framework parameters that influence application performance. Finally, we look towards how such lessons can be applied to the ever-changing Big Data landscape.

Chapter 2

Leveraging Temperature to Schedule MapReduce Jobs for Energy Efficiency

Author's note: The text in this chapter is largely from my conference paper "Configuring a MapReduce Framework for Dynamic and Efficient Energy Adaptation" [31] and my journal paper "MapReduce Framework Energy Adaptation via Temperature Awareness" [30]

MapReduce has become a popular framework for Big Data applications. While MapReduce has received much praise for its scalability and efficiency, it has not been thoroughly evaluated for power consumption. In this chapter, we will explore the possibility of scheduling in a power-efficient manner without the need for expensive power monitors on every node. We begin by considering that no cluster is truly homogeneous with respect to energy consumption. From there we

develop a MapReduce framework that can evaluate the current status of each node and dynamically react to estimated power usage. In so doing, we shift power consumption toward more energy efficient nodes which are currently consuming less power. This model shows that given an ideal framework configuration, certain nodes may consume only 62.3% of the dynamic power they consumed when the same framework was configured as it would be in a traditional MapReduce implementation.

2.1 Introduction

MapReduce was originally designed for a cluster of commodity machines [19], and popular implementations such as Hadoop[3] see better performance in a homogeneous environment. Many frameworks assume that workers complete their jobs at the same time and with approximately the same cost per node. As not all clusters are homogeneous, Hadoop has a "straggler" mechanism through which they preemptively re-schedule jobs that were assigned to nodes that are not producing results as quickly as other nodes. Related work [81, 76] shows this mechanism to be insufficient in heterogeneous clusters, as it struggles to balance the workload when there are enough slow nodes in the cluster.

When considering energy efficiency we find that a cluster cannot be completely homogeneous. Machines have various idiosyncrasies, including but not limited to: amount of thermal paste on the processor, fan speed, fan size, location in proximity to a cooling unit, and heat sync efficiency. These variations mean that some machines may produce higher temperatures than others even when performing the same work on the same data. This requires that some machines receive additional

cooling compared to another machine with the same specifications. This translates to a larger energy demand in machines that run at higher CPU temperatures. In the converse, we know that performing additional computations and data loads requires additional work; in turn requiring additional power. Meaning that a hotter machine is likely to be doing more work than a cooler machine. A motivating factor in this work is that our homogeneous cluster is heterogeneous with respect to power consumption and in order to combat this we need a MapReduce framework that can dynamically schedule work based upon power consumption of an individual node.

In this chapter, we will quantify the relationship between CPU temperature and energy consumption, and show that CPU temperature is a reliable indicator of current power consumption with respect to a single worker node. Utilizing this, we design and implement a MapReduce framework that dynamically schedules jobs using CPU temperature as a metric via which we can estimate power consumption. We will test various aspects of our framework for their impact on energy consumption and show that through scheduling we can reduce the amount of additional power needed by 37.7% on individual nodes when compared to the same framework utilizing methods typical of other MapReduce implementations.

2.2 Preliminary Findings

In order to schedule for energy, we need to find a way to quantify the energy usage of a node in a relatively efficient manner. To that end, we need to find a metric through which there is a strong correlation to power consumption. Our goal is to find a measurement that would not result in a need to affix external hardware

monitors (such as power meters) to each of the nodes, as this is not cost effective and not practical for large clusters. We consider CPU temperature as such a metric.

To determine whether or not CPU temperature is a viable metric for our purposes, we needed to test for a reasonable correlation between CPU temperature and energy consumption. On the surface we feel that these two measurements should be correlated since more work requires more power, and work generates heat on the chip. In order to test this assumption we designed several experiments. These experiments were carried out on a machine with the following configuration:

- Intel Xeon CPU E5320 @ 1.86GHz
- 8MB L2 Cache
- 64-bit Linux 2.6.32

We ran tests utilizing the Great Internet Mersenne Prime Search program *mprime* [29]. The aspect of *mprime* that we relied on for this testing was the torture testing. The torture tests stress the system in three different ways, as per the program documentation.

- **Test 1:** Stresses the FPU with minimal testing of RAM as all data fits in the L2 cache.
- **Test 2:** Stresses the FPU and some RAM, consumes maximum power, produces maximum heat.
- **Test 3:** A combination of tests 1 and 2, that balances the type of stress between resources, stressing the FPU and lots of RAM.

It was necessary to stress these various components of the system as we are aware that there is heterogeneity amongst MapReduce workloads. We know that some workloads are I/O intensive (e.g. CloudBurst) [63], and some are CPU intensive (e.g. Matrix Multiply), as per [75]. In addition to using these three *mprime* torture tests as a reference, we developed three different methods of gathering data to help simulate variability within a given workload.

- **Stress:** Iterate through a loop while preforming floating point operations. We take 1000 total readings of CPU temperature and system power upon hitting various checkpoints in the loop. This simulates an intensive workload, especially when run in conjunction with *mprime* torture testing.

- **Temp:** Iterate through a loop and take 100 measurements of both CPU temperature and system power, pausing briefly between readings to allow the temperature to drop back down. This simulates running a variable intensity workload.

- **Simple:** Iterate through a loop and take 100 measurements of both temperature and power while generating as little additional work as possible.

The results of our tests are shown in Figure 2.1 and help us to identify a correlation between temperature and power consumed on an individual node. The bar labeled *test1* shows a Pearson correlation coefficient of 0.711 when considering all data gathered running Test 1 for *mprime* and all three methods of gathering data. Similarly, the bars labeled *test2* and *test3* show a Pearson correlation coefficient of 0.50 and 0.81 on all data gathered while running Test 2 and Test 3 for *mprime* respectively. The bar labeled *simple* shows a Pearson correlation coefficient of 0.36

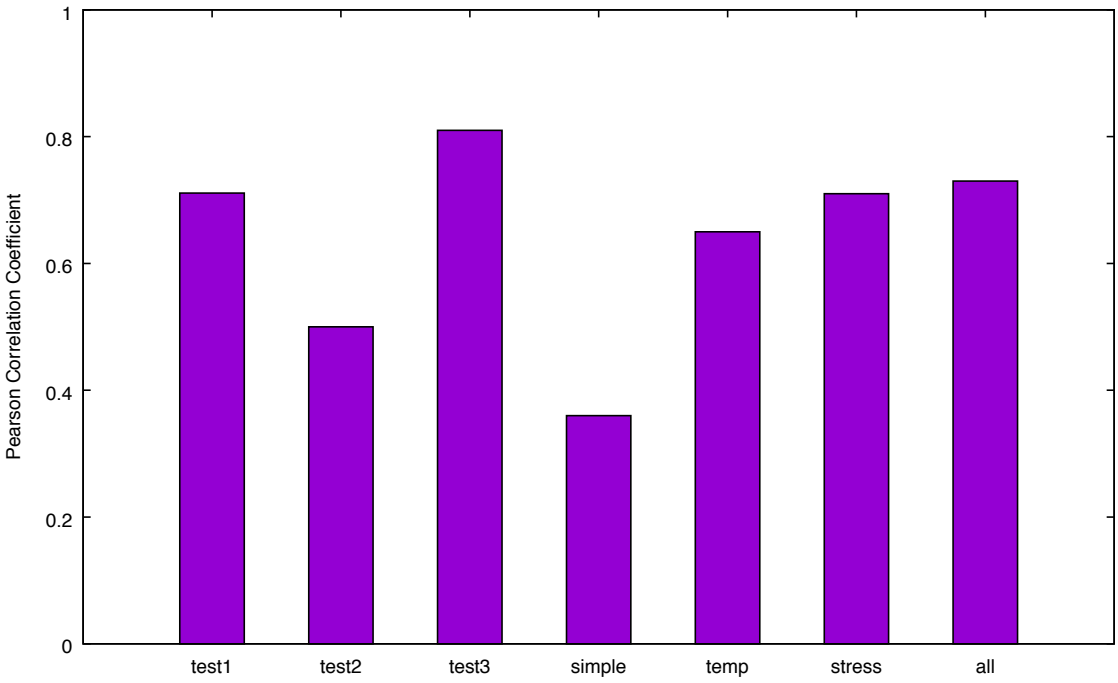

Figure 2.1: The Pearson Correlation Coefficient of categorizations of the combined data from individual tests.

when considering all data collected running the Simple gathering policy. Similarly, the bars labeled *temp* and *stress* show a Pearson correlation coefficient of 0.65 and 0.71 on all data generated using the Temp and Stress methods of gathering data respectively. Finally, the bar labeled *all* represents the Pearson correlation coefficient of all data when considered as a whole, with a value of 0.73. From these tests, we can see that collecting data in the simple fashion, and running *mprime* with Test 2 shows the least correlation between power and temperature. This is not a result we had anticipated on the whole, but after considering this fact, we derived the following reasons that the correlation may be weakened:

- The cooling system in place on an individual machine requires power to operate, and by expelling additional power, the temperature of the CPU may drop, thus having the opposite of the effect we first anticipated.
- External cooling systems (i.e. air conditioning, and neighboring computer's cooling systems) require no power through the meter, and yet drop the CPU temperature, thus having the effect of reducing temperature while power consumption may remain the same.
- When a node's work comes to completion, the power consumption is immediately reduced as the CPU is executing many less commands. However, the higher temperature that was generated as a result of the workload takes time to dissipate, thus further weakening the correlation between power and CPU temperature.

These three justifications begin to explain why it is that the correlation between

power consumption and CPU temperature is not as strong as was initially believed. We find that the data from our experiments indicates that there is a positive relationship between power consumption and CPU temperature; as the CPU temperature goes up, so does the power consumed. We also find that the relationship between power and temperature grows stronger when we stress the node more while taking readings. We determined that with a correlation coefficient of 0.734, our results have confirmed our hypothesis that CPU temperature and power consumption are related. To that end, we will use CPU temperature in an effort to schedule jobs on worker nodes. Note that in the future we can use RAPL [60] measurements to determine the power consumption of the CPU. However, using these preliminary findings, we worked to design a MapReduce framework that would allow us to more intelligently schedule jobs on worker nodes around power consumption.

2.3 Design

Our design is based upon previous work with MARIANE [27] and MARLA [28], both utilize a shared filesystem to distribute jobs across the network. Works such as GreenHDFS [38] and the contributions of Chen et al. [16] indicate that an energy-related weakness of HDFS is the way in which work is distributed and replicated. While distribution of jobs based upon data placement and replication makes Hadoop the de facto standard for MapReduce implementations, we believe that this strong coupling between implementation and filesystem has the potential to inhibit energy efficiency. As such, the flexibility of choosing the most energy efficient Distributed Filesystem (DFS) currently available was a must-have for our

framework.

We also approach this problem with the realization that even the most homogeneous clusters have some elements of heterogeneity, especially with respect to energy efficiency. This is due to the fact that aside from the traditional specifications reported for a node there are many characteristics of a node that affect a node's temperature as well as its energy efficiency. These characteristics include, but are not limited to: proximity to external cooling and heating elements, amount of dust in the case, fan speed and size, amount and distribution of thermal paste on the chip, and efficiency of the heat sync. Independently of this work, related work by Li et al. also demonstrates that given the same type of nodes, with the same utilization rates, and frequency, that the temperature differences can also change the energy footprint [44]. Because of this realization, another feature we wanted in our framework was the ability to react to such heterogeneity. In order to do this, we borrow aspects from MARLA and allocate more than one job per node. This way, the faster (and/or more energy efficient nodes) can take on additional tasks and leave the slower (and/or less efficient nodes) to cool off and consume less power.

In order to meet these design goals we begin with an NFS distributed filesystem. Following this, we have set up job allocation in the following way:

1. Split the input into a user-defined number of tasks. In later chapters we will perform additional testing to determine the optimal number of splits without having to necessarily rely on user discretion.

2. Perform the following simultaneously:

   (a) Distribute one job to each node in the cluster.

(b) The master node starts a thread that notes each node's temperature to see if it is above a certain threshold. Currently the temperature threshold is also user-defined. It may be necessary in the future to set up individual node thresholds because some chipsets have a higher operating temperature than others.

3. Once a node has completed its first task, if its temperature is below the user-defined boundary temperature, we allocate another task to that node. However, if the temperature exceeds the boundary temperature, this node simply waits until either its temperature is low enough to run another task, or the job completes.

This process repeats as long as there is work to be done. Once all work has been assigned, the fault tolerance module begins work and distributes jobs without regard to temperature of a node. Note that this trade-off is made to preserve fault tolerance; without it a task may get stuck waiting for nodes to cool down and the job may never finish. Since there are user-defined parameters in this implementation, we will first discuss the impact of each of these parameters, beginning with the boundary temperature, then moving on to number of tasks assigned to the cluster.

2.4 Implementation

We designed our MapReduce framework so as to exploit the implicit heterogeneity of some MapReduce clusters, and in order to accomplish this we relied on two user-defined parameters. The first such parameter is the boundary temperature,

which is used to determine the temperature at which a node should no longer be considered for rescheduling. The second such parameter is the number of tasks that are created from a single MapReduce job; with each task corresponding to a subset of the input file. User-defined parameters lend to variability in our MapReduce implementation so we will first discuss the impact of each of these parameters, beginning with the boundary temperature, then moving on to number of tasks assigned to the cluster.

2.5 Experimental Setup

We collect power data for one node in our cluster using a Watts Up? .Net power meter. We do this as we make local decisions regarding energy consumption and so as few as one node may have any power consumption changes. In an effort to determine the actual realized power savings, we run tests designed so that the single node takes on opposite sides of saving and consuming more energy. As per discussions in related work [17, 16] the energy level of the master node is not measured, since the master node contributes approximately the same amount of energy to the cluster, regardless of the cluster's size. These works also do not report power consumption on behalf of the network switch because when a cluster is not isolated such results could corrupt the data if other machines are utilizing the network.

The cluster is set up with our master node having the following specifications:

- Intel Xeon CPU 5150 @ 2.66GHz
- 4MB L2 Cache

- 64-bit Linux 2.6.32

and with our worker nodes having the following specifications:

- Intel Xeon CPU 5150 @ 2.66GHz
- 4MB L2 Cache
- 64-bit Linux 2.6.32

with the exception of our metered worker node, having the following specifications:

- Intel Xeon CPU E5320 @ 1.86GHz
- 8MB L2 Cache
- 64-bit Linux 2.6.32

Additionally, all nodes in the cluster have the *lm-sensors* [53] package installed in order to be able to determine the CPU temperature of each node. It is important to note that although 80% of the worker nodes have the same configuration, two of the non-metered worker nodes ran with a higher average CPU temperature than the others. One such node had a tendency to run just over 100°C, and the other such node ran just over 115°C. The data is shared between nodes using NFS, hosted on a local server, but the framework makes no assumptions regarding the setup and any shared filesystem could be used. Looking forward, this kind of flexibility is necessary as research in the area of green distributed file systems is in progress, see [39, 84]. As breakthroughs are made, our system must be adaptable and able to realize the changes necessary to find power savings. Each of our experiments used

the traditional WordCount application. The average of all iterations of a given experiment is reported.

2.6 Parameter Exploration

In this section we describe the results of the experiments performed on the cluster as described in Section 2.4.

2.6.1 Boundary Temperature Sensitivity

Based upon the way our framework is designed, the temperature that is used to decide whether or not a node is able to take on more work is a parameter that can limit performance. In light of this, we perform all tests with three different temperatures as our boundary temperature. As was described in Section 2.4, 20% of the worker nodes had a tendency to run at a higher temperature than the other 80%. As such, the boundary temperatures we selected were 80, 90, 100, 110, 120, and 130°C. However, for the sake of brevity, we will only discuss data from 80, 110, and 120°C as they correspond to 60%, 80% and 100% reschedulable workers and the results for 90 and 100°C, mimic those of 80°C, similarly for 120 and 130°C.

The depiction of some of these experiments is shown in Figure 2.2. The graphs in this figure all have Execution Time in seconds on the Y-axis and File Size in MB on the X-axis. The first graph displays the data from tests executed when the number of tasks is equal to the number of workers. The second graph displays the data from tests executed when the number of tasks is two times the number of workers. The final graph displays the data from the tests executed when the number

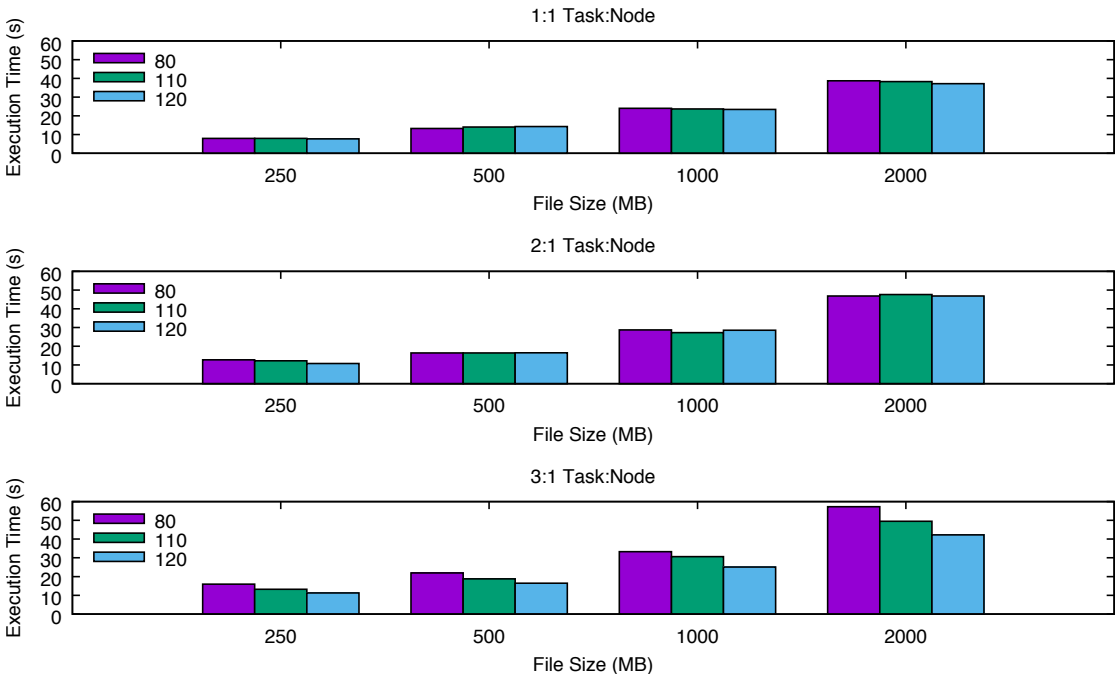

Figure 2.2: Execution time comparison of Boundary Temperature settings relative to the ratio of the number of tasks to worker nodes.

of tasks is three times the number of workers. The graphs show that when the temperature is considered in scheduling additional tasks, as the boundary temperature decreases, the execution time increases in nearly all cases. This is because when the temperature of a node exceeds the boundary temperature, the node is considered to be temporarily dead.

A boundary temperature that is set too close to the idle temperature of the CPUs in the cluster could potentially make it such that very few nodes are eligible for obtaining a second task after completing their first task. This also means that there is potential for a live-lock to occur. In order to eliminate this possibility, some knowledge of the cluster to which this framework is deployed can provide a guaranteed means of preventing such conditions. Ideally, if the idle temperature of all nodes was known prior to deployment then the boundary temperature can be set so that there is an ample temperature range between an active node and an inactive node. Further evidence of the importance of properly setting user-defined variables, is seen in the increase in execution time as boundary temperature decreases. We can also see that as we reduce the ratio of tasks to worker nodes the effects of changes to the boundary temperature are reduced. Note these trends displayed in the second and third graphs of Figure 2.2. In the third graph, there are three times as many tasks as there are workers, and the improvement in turnaround time increases as the file size increases. Comparing this to the second graph, where there are only twice as many tasks as there are workers, the improvement in turnaround time over the data points in the first graph are minimal at best, and there is not a marked improvement in turnaround time as file size increases.

Figure 2.3 shows graphs that have the total change in watts of power consumed

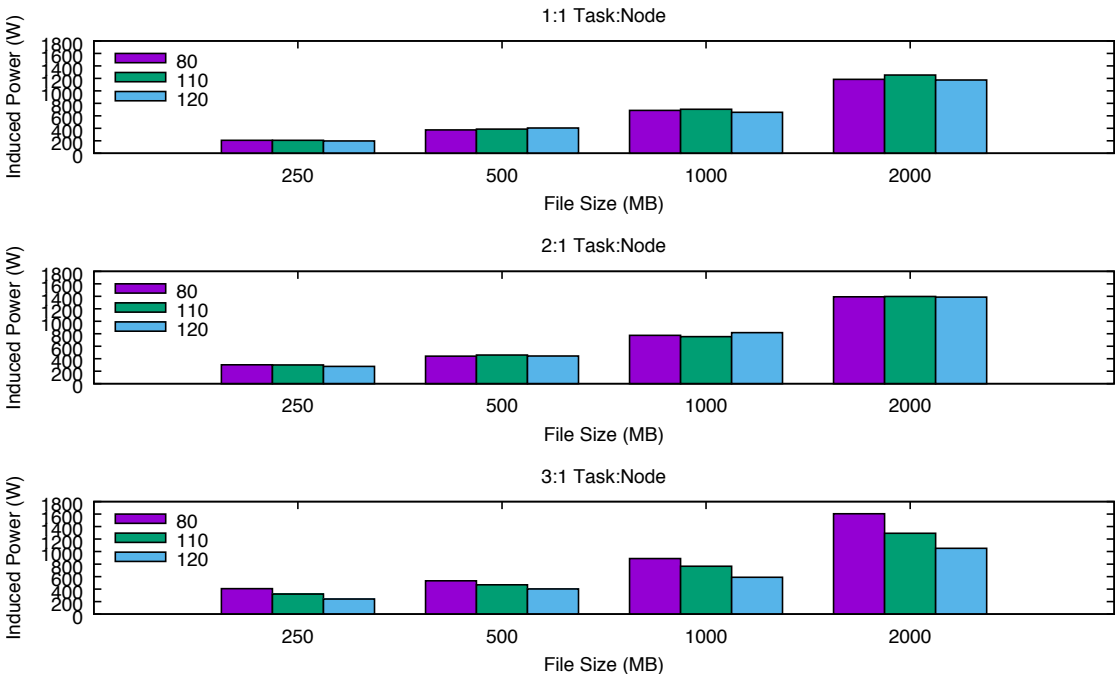

Figure 2.3: Induced power comparison of Boundary Temperature settings relative to the ratio of the number of tasks per worker node.

on the Y-axis and File Size in Bytes on the X-axis. Figure A displays the data from the tests executed when the number of tasks is 3 times the number of workers. Figure B displays the data of tests executed when the number of tasks is 2 times the number of workers. These graphs show the exact same trend as the turnaround time graphs in 2.2. This means that fine-tuning the boundary temperature to the cluster this framework will be deployed on can save a significant portion of power. Note that these trends are not absolute. In our experiments the node that we tested for power consumption did not run hot, but was a slow node. As such, it executed jobs when the temperature was too high on the hotter nodes, despite it being a slow node. Consequently, this node was not chosen when there were a smaller number of available jobs and cooler, faster nodes could accept them instead. In order to illustrate this we will look at Figure 2.4.

This graph has the total change in watts of energy consumed on the Y-axis and the number of jobs available with respect to the number of worker nodes in the cluster on the X-axis. This graph presents these data points and separates them out by the boundary temperature at the time the data points were recorded. Note that this figure indicates that the power consumed varies quite a bit based upon how many jobs are spawned and what percentage of the nodes will be deactivated after their first job. Now we will turn our attention toward the splitting of data into jobs based upon the number of nodes in the cluster.

2.6.2 Data Split Sensitivity

Observe Figure 2.4, it can be determined that part of our heterogeneous cluster's success at processing MapReduce data quickly is dependent on how well we split

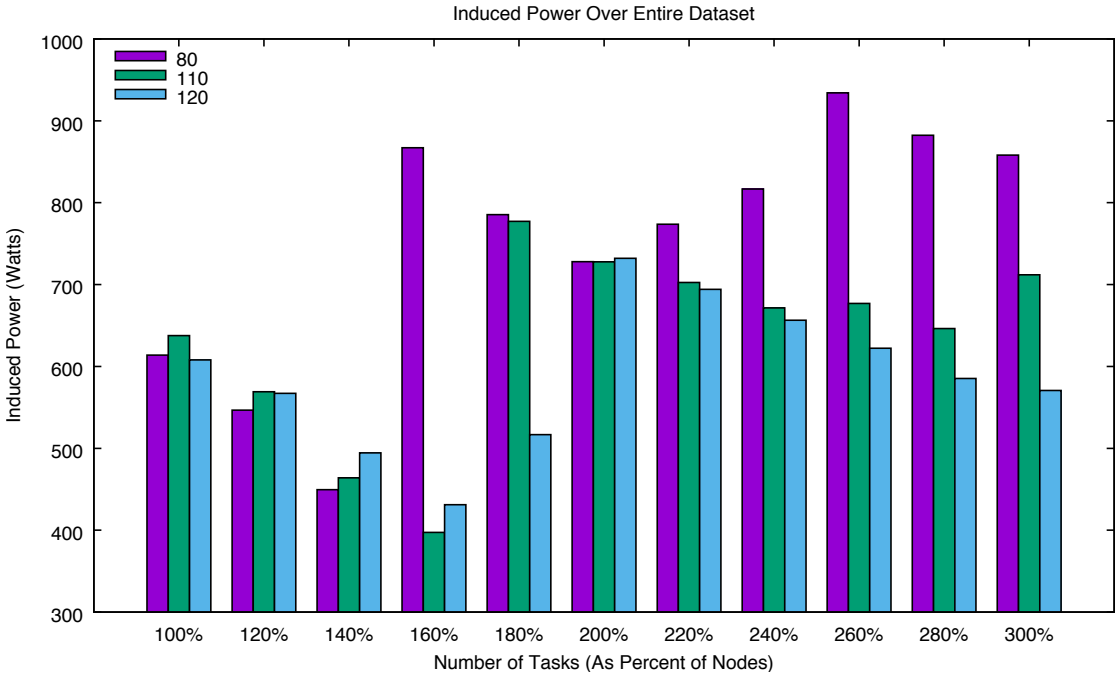

Figure 2.4: Induced power for each configured Boundary Temperature while increasing the number of tasks for a given job.

our data. To further analyze this we will consider how much the file size depends on how many pieces we can split it into. Since there is some amount of overhead associated with stopping and starting a MapReduce job in any framework, we will consider how the splitting of various size files into multiple pieces impacts our overall throughput. In 2.5 we see data for execution time based upon number of splits for a 500MB and a 1GB file. While this same test was performed on a file of 250MB and 2GB as well, this data is omitted because the 250MB file has a similar fit to the 500MB file, and the 2GB file has a similar fit to the 1GB file. Instead we will analyze the 500GB and 1GB files, and we will discuss their similarities and differences. The graph in Figure 2.5 has an X-axis that represents the number of jobs as a percentage of the nodes in the cluster, and a Y-axis that represents the execution time of the MapReduce job in seconds. There is one dataset for each temperature threshold that the scheduling was based on. This graph is from a file of 500MB. The graph in Figure 2.6 is the same that in Figure 2.5 except that it is for a file of 1GB.

Both graphs in Figure 2.5 and 2.6 show two intersecting points, the first one is at a task to node ratio of one. This value represents when there are exactly as many tasks as there are nodes, which is the traditional ratio used by MapReduce frameworks. The next point of intersection is at a task/node ratio of two. This intersection was predictable, as it is probable that the delay between the fastest and slowest nodes finishing jobs is less than the time it takes to execute the task, thus resulting in each node getting two tasks.

An interesting observation is that when the number of tasks is three times the number of nodes, we do not see an intersection point. One possible explanation

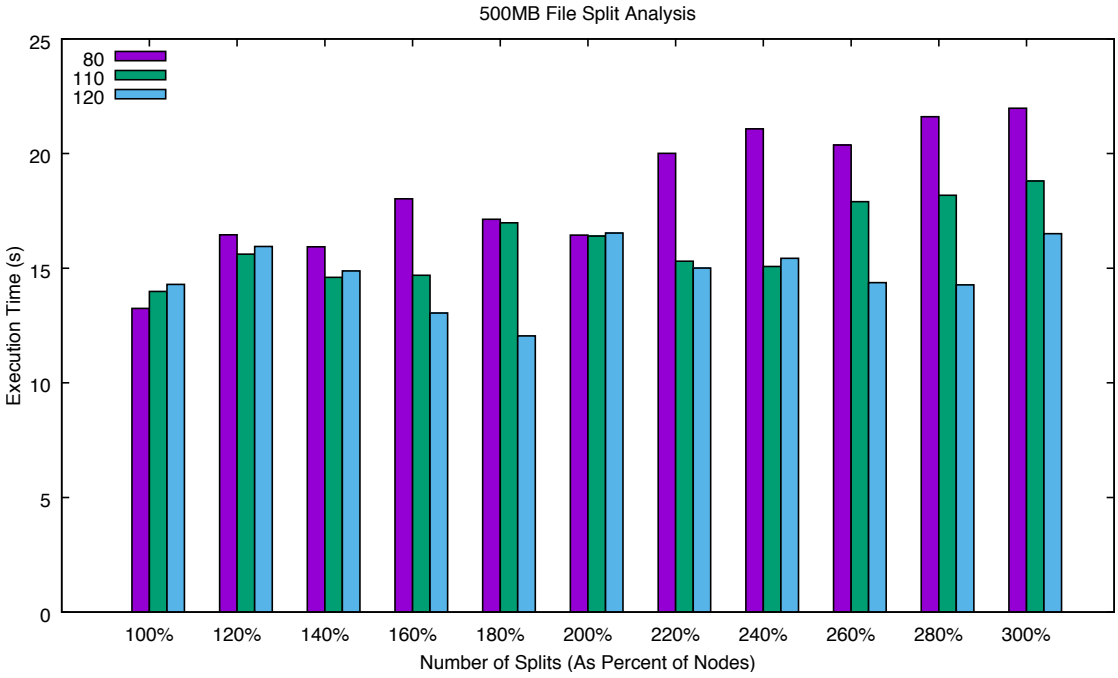

Figure 2.5: Execution time of 500MB file relative to the number of file splits, per boundary temperature

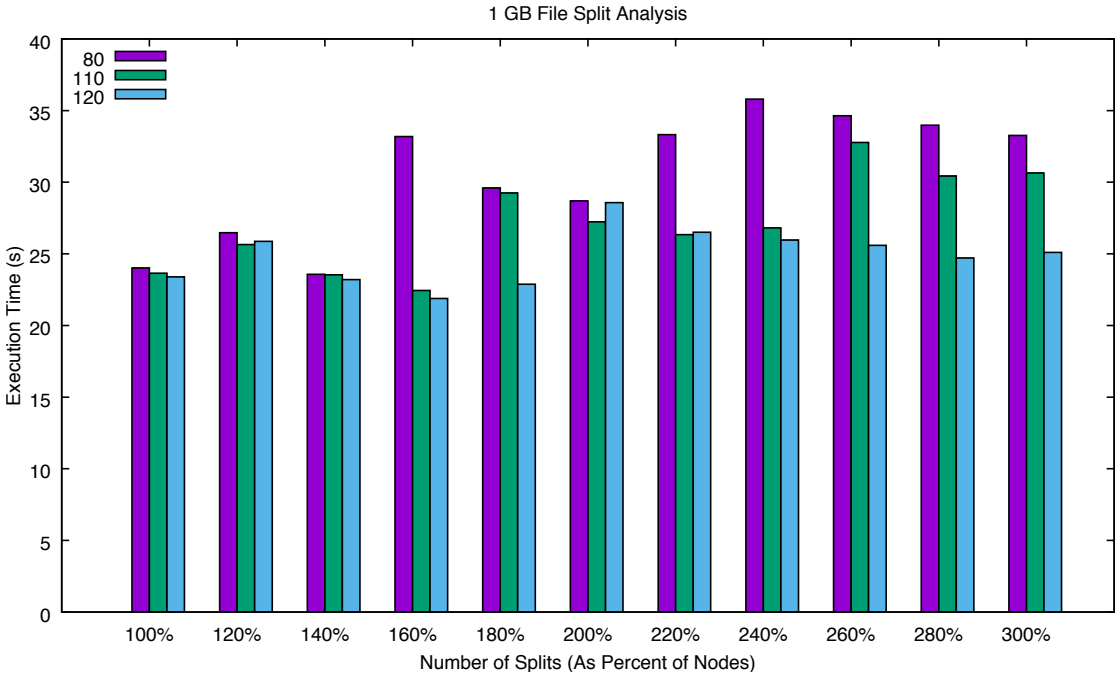

Figure 2.6: Execution time of 1GB file relative to the number of file splits, per boundary temperature

for this can be seen when comparing this to the intersection that occurred at the task/node ratio of two. At this point we saw that each node was given two tasks to complete and we determined that this likely happened as the delay between the fastest and slowest nodes completing a job is less than the the time taken to execute one job. However, at the task/node ratio of three, the tasks are shorter because there are more of them, and it is now more likely that the delay between the fastest and slowest nodes completing a job is more than it takes to execute one job. It is important to note here that our cluster is only moderately heterogeneous, and that more heterogeneity would show a vast change in the effectiveness of a given task:node ratio.

We can also see from both sets of graphs that the threshold temperature value is a contributing factor to execution time as the number of splits increases. This makes sense as a MapReduce job can complete only as fast as the slowest node completes the job. For this reason a decrease in the boundary temperature results in an increase in the execution time. Note that when the boundary is low enough, several nodes are effectively removed from the cluster and so the remaining nodes become slowed down by having to do a larger percentage of the work. However, our cluster is only moderately heterogeneous and we hope that with a larger, more heterogeneous setup, lower boundary temperatures may not as dramatically shift execution time.

One feature that defines each of these graphs is the relative smoothness of each of the temperature plots. In the 500MB graph, the 110°C plot has the least variation in the slope of its various segments, whereas in the 1GB graph, the 120°C plot has the least variation in the slope of its segments. The sum of the variance

in the slopes of all segments in the 500MB file is 194.5, whereas the same value for the 1GB file experiments is 731.9. This means that as the input size grew by a factor of two, the variance grew by 376.3%, nearly a factor of four. With respect to the 500MB file experiments, we see that our results are more consistent (independent of the number of file splits) if we leave some of our thermally excited nodes out of rescheduling. The data from the 1GB file experiments tells us that as file size increases the number of splits becomes increasingly important. The results of subsection 2.6.1 together with subsection 2.6.2 indicate that as the number of tasks increases and the data size grows the framework becomes less dependent on the boundary temperature, indicating that the framework is scalable.

2.6.3 Power Savings

As was discussed in Section 2.4 for our experiments we collected data on a single worker node as we made local decisions regarding job scheduling and hope that such decisions will translate to global power savings. While our present experiments do not consider the power consumption of the entire cluster, these experiments serve to act as a proof of concept that our decisions regarding temperature adjust the temperature and power consumption of individual worker nodes, with cluster power savings being quantified at a later date.

Consider the results of our power analysis on a single node as presented in Figures 2.7, 2.8, 2.9, and 2.10. Each of these figures has the total change in watts of power consumed on the Y-axis and the number of tasks available with respect to the number of worker nodes in the cluster on the X-axis. The data for these graphs varies based on file sizes of 250MB, 500MB, 1GB, and 2GB respectively. The data

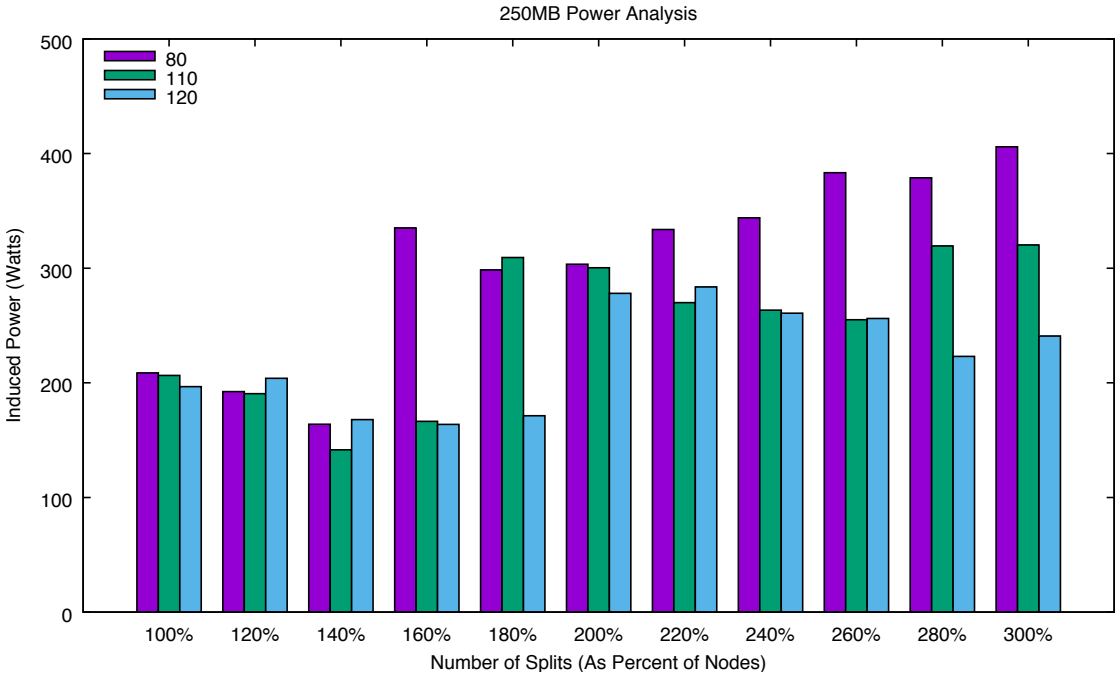

Figure 2.7: Induced power per boundary temperature relative to the number of splits for a file of size 250MB.

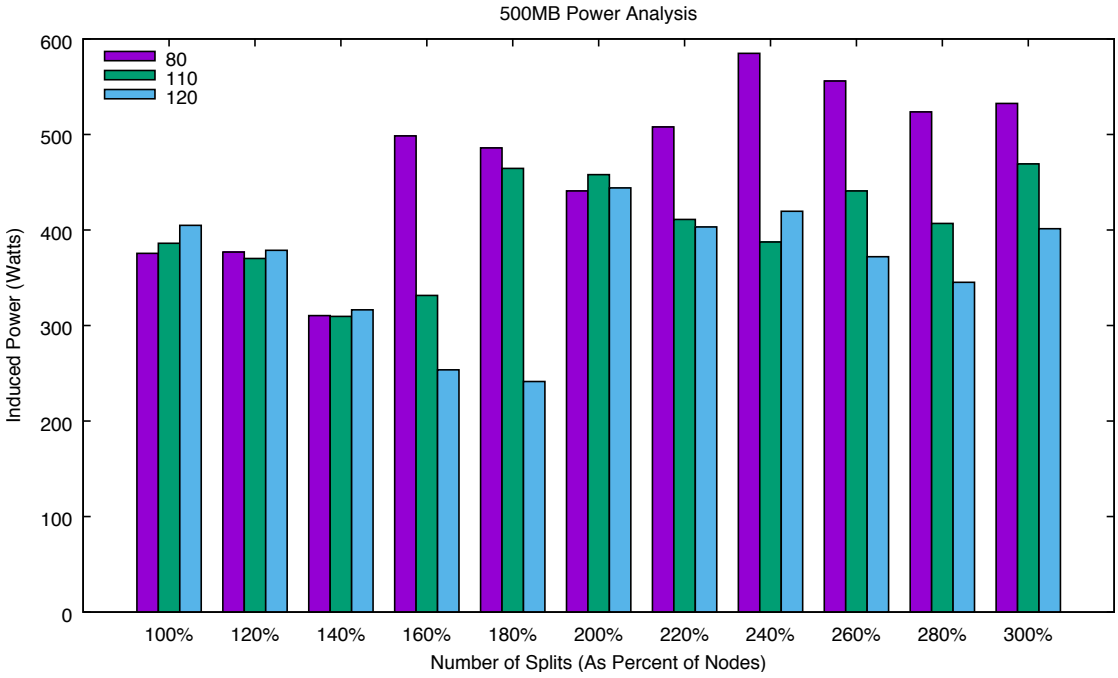

Figure 2.8: Induced power per boundary temperature relative to the number of splits for a file of size 500MB.

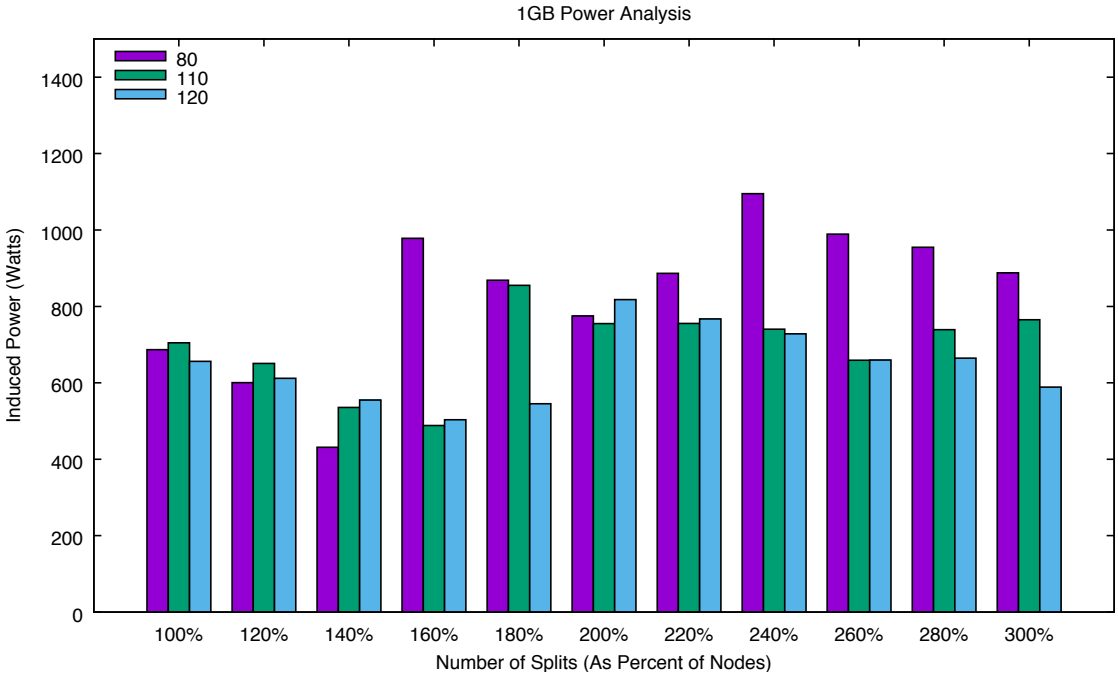

Figure 2.9: Induced power per boundary temperature relative to the number of splits for a file of size 1GBB.

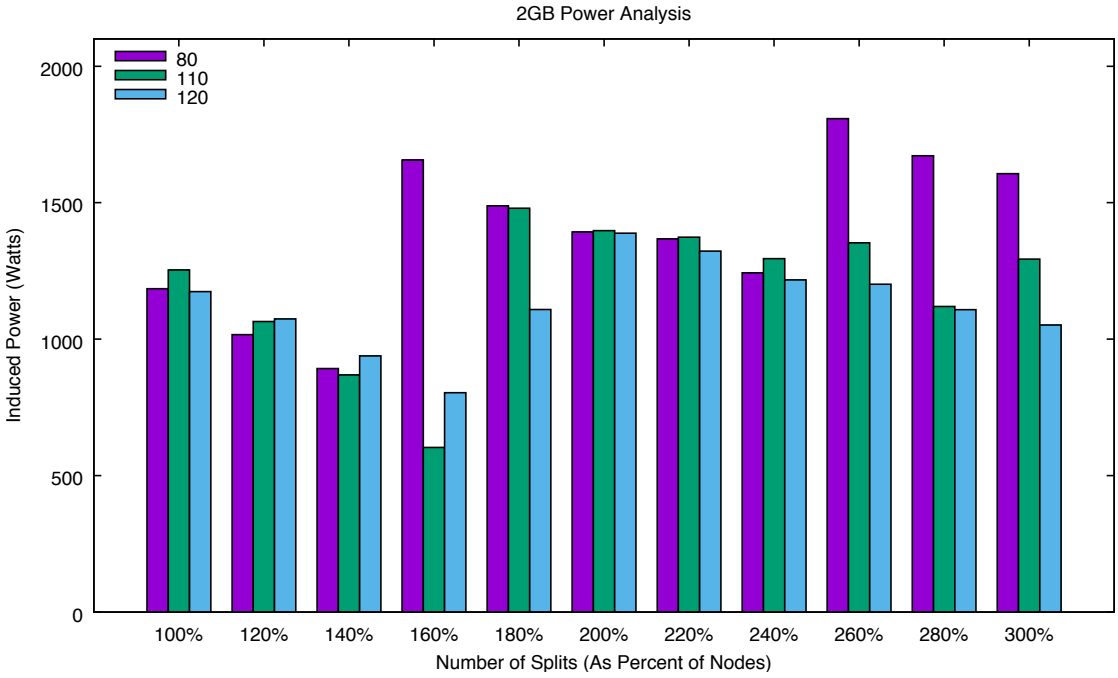

Figure 2.10: Induced power per boundary temperature relative to the number of splits for a file of size 2GB.

is also separated by the boundary temperature at the time the data points were recorded. We consider only the additional power (total power - idle power) for reasons discussed by [11].

As with several other trends, we see that both the boundary temperature and the number of splits play an important role in the realized energy savings. We can see from this node, that as it is not used for rescheduling when the number of jobs is low, power is saved over the traditional one task per node approach presented in MARIANE [27]. We see in the 2GB file case that when the boundary temperature is 80°C there are two peaks of power consumption, and these peaks correspond to when the node is rescheduled once and twice. When the node is not rescheduled and the tasks become shorter (the number of splits increases) our power consumption demands decrease. We see similar trends in the other boundary temperatures as well, where the data reaches two local minimums and two local maximums, with the minimums located just before an extra task is scheduled, and the maximums occurring just after.Note that as the number of tasks increases, the disparity between the minimum and maximum is decreased.

Another noteworthy trend is that compared to the data using the number of splits and the execution time, more data points have similar values amongst the various temperature boundaries when change in power is considered, especially as the file size increases. In the 250MB example, there are 3 cases where the range of the values graphed at each temperature falls less than 10% of the average of those values; this is true of both execution time and power. In the 2GB example, there are 4 cases where the range of the values graphed at each temperature falls less than 10% of the average of those values with respect to execution time;

where there are 6 such cases with respect to Watts, an 18.18% increase in similar values. This trend confirms that boundary temperature does effect the job's execution time, it effects the single node's power consumption only when the boundary temperature forces the node to take on additional tasks. These results indicate that there is some balance achievable between power savings and execution time by adjusting the boundary temperature, namely that even though execution time varies with boundary temperature, we see that the power consumption is 18.18% more consistent.

2.7 Comparison With MARLA

As performance is necessary for any successful MapReduce framework, we will take this section to discuss the effects that scheduling for energy awareness has on performance. In our view, some performance loss is acceptable if there is an energy savings. However, as was pointed out in other work [42] the longer a machine is active, the more energy it consumes, so performance still remains a priority. In this section we will discuss the performance impacts of setting various elements of our framework, comparing our framework to another framework also based upon MARIANE[27], MARLA[28]. The primary difference between our framework and MARLA is that we have scheduled this framework for energy awareness, whereas MARLA schedules for performance alone. Comparing our framework to MARLA is sufficient since MARLA has been compared to Hadoop and MARIANE; showing improved performance over these other frameworks in heterogeneous clusters.

Figure 2.11 has the number of splits as a percentage of nodes in the cluster on

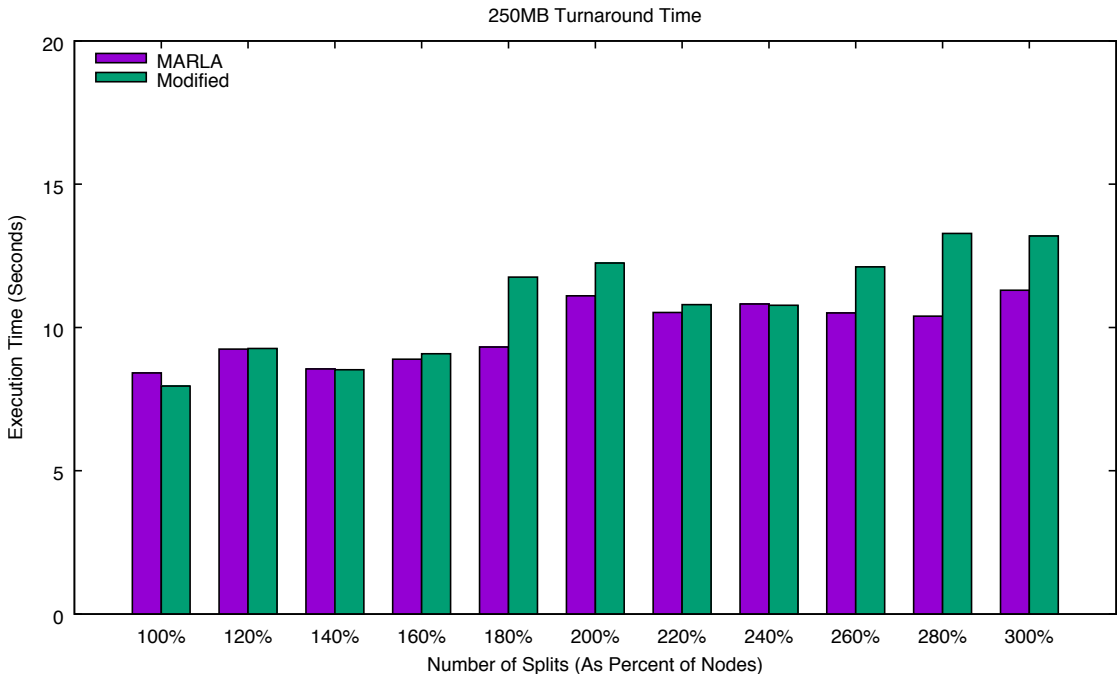

Figure 2.11: Execution time with respect to the number of splits of a 250MB, and a boundary temperature of 110°

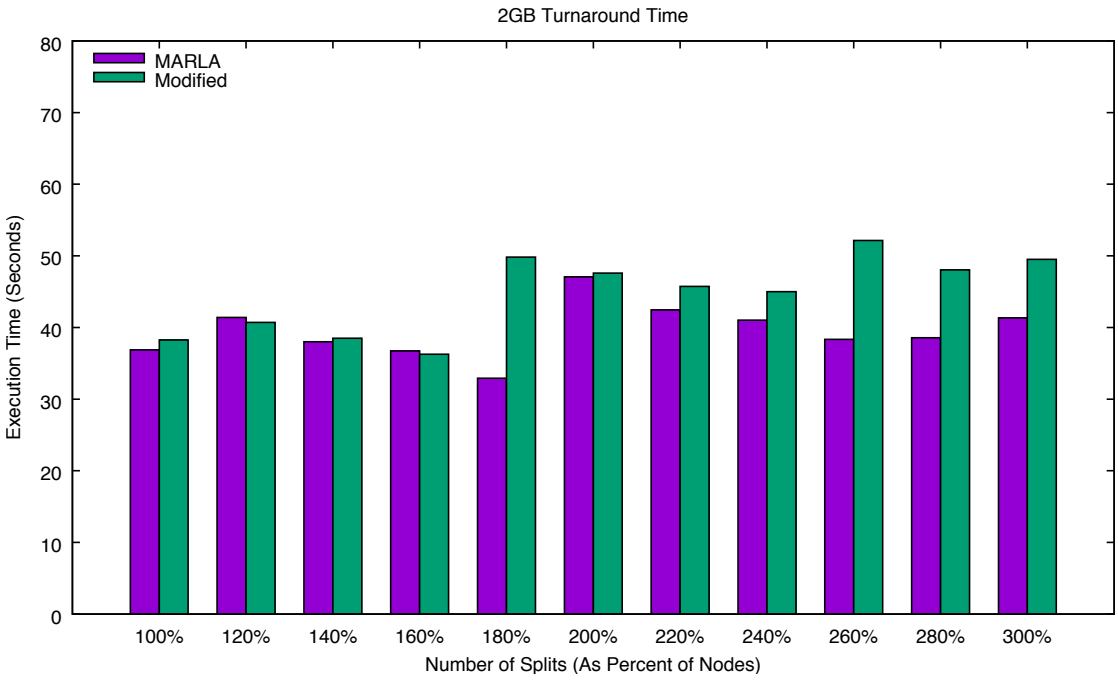

Figure 2.12: Execution time with respect to the number of splits of a 2GB, and a boundary temperature of 110°

the X-axis, and the execution time of the job in seconds on the Y-axis. It also displays the results for MARLA and our framework, a modified version of MARLA, with a boundary of 110°C at the 250MB file size level. Similarly, Figure 2.12 plots the same information except with respect to the 2GB file size level.

Recall from Section 2.2 that a framework that works well in a heterogeneous environment will be better suited to energy adaptive scheduling, as no cluster is truly homogeneous for scheduling. Our results in Figures 2.11 and 2.12 indicate that our only performance loss over MARLA occurs when the number of splits and the runtime length of each split precludes some nodes from being rescheduled due to their temperature variations. For this reason, we can see why it takes remarkably longer for our framework to complete in the window with the number of tasks between 160 and 200% the number of nodes; as well as after 240% where the runtimes begin to diverge. Recall from the previous section that this same scenario also changes the power consumption of an individual node. This is an expected result, but it shows that if our framework is properly tuned to have the appropriate boundary temperature for the given heterogeneity of the cluster, we can realize times on par with those discussed in MARLA [28]. Since we know that our execution times are similar to those that we gathered using MARLA, we can see that some power savings on the cluster level may be achieved. The graphs in Figures 2.13 and 2.14 have the number of splits as a percentage of nodes in the cluster on the X-axis, and the change in power consumption in watts on the Y-axis. The graph displays results for MARLA and our framework, a modified version of MARLA, with a boundary of 110°C at the 250MB and 2GB file size level respectively.

We saw that the results of Figures 2.11 and 2.12 indicate that the turnaround

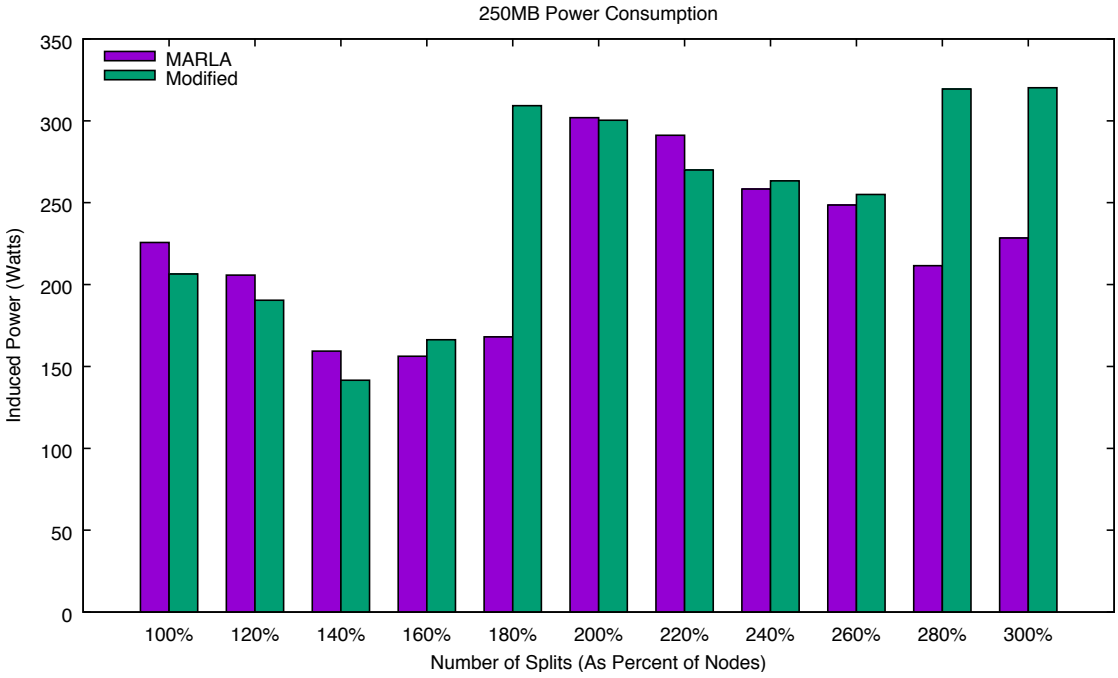

Figure 2.13: Execution time with respect to the number of splits of a 250MB file, and a boundary temperature of 110°, compared to MARLA

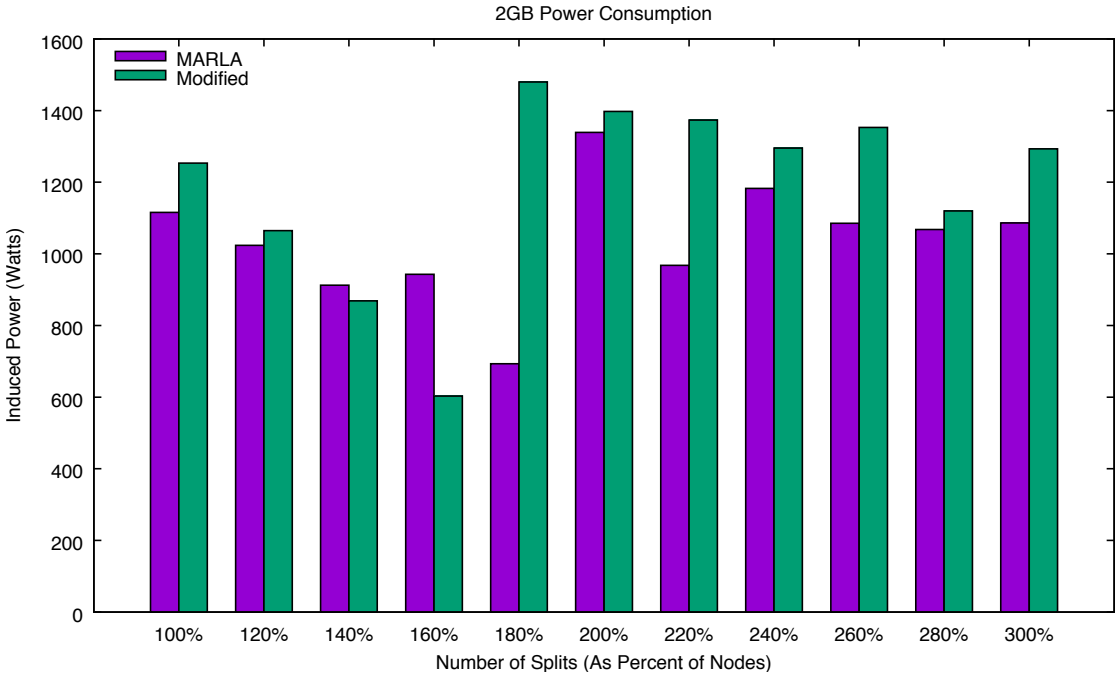

Figure 2.14: Execution time with respect to the number of splits of a 2GB file, and a boundary temperature of 110°, compared to MARLA

times of our framework are similar in many instances to those of MARLA. After noting this, we decided that if we realized power savings on a single node between our framework and MARLA, we should be able to realize power savings throughout the cluster given the right configuration of our framework. We present our results in Figures 2.13 and 2.14. When the file size is small, e.g. in the case of the 250MB file size, our power consumption closely follows the trends and values of the unmodified MARLA framework except that the changes in power consumption occur in our framework before they do in MARLA as we eliminate one node from the cluster since our boundary temperature in this instance was 110°C. However, when the file size is large, e.g. in the case of a 2GB file size, we see a much smoother trend of power consumption in our framework than we do in MARLA. While our framework continues to attain its minimum and maximum power consumption with a smaller number of tasks than MARLA, we see that some power savings are realized and that our framework provides a method for scheduling MapReduce applications for energy.

2.8 Conclusions

In this chapter we designed and implemented a MapReduce framework whose scheduling is dynamic and energy aware. Offering the following contributions:

- Establish a positive correlation between CPU temperature and power consumption, and we used this correlation to try and assign jobs to nodes that are not as hot as other jobs.
- Design and implement a MapReduce framework that is able to utilize the

correlation between power and CPU temperature.

- Test various user-defined characteristics of our framework in an effort to determine the effect each of them has on the success of our framework with respect to both turnaround time of the job and power consumption of an individual node.

- Show potential for a MapReduce framework that can schedule in an energy-aware manner without having to rely on expensive power hardware attached to each node.

Chapter 3

Considerations for Combating Performance Heterogeneity

Author's note: The text in this chapter is largely from my conference paper "Configuring A MapReduce Framework for Performance-Heterogeneous Clusters" [32] and my journal paper "Performance Analysis of Adapting a MapReduce Framework to Dynamically Accommodate Heterogeneity" [33].

3.1 Design

3.1.1 Deferred Binding of Tasks

We wish to characterize the performance of delayed mapping of data and tasks to worker nodes in the MARLA MapReduce framework. This section describes important MARLA features and distinguishes MARLA from Hadoop, primarily with respect to how the two frameworks operate on performance heterogeneous

clusters. We have described MARLA in more detail elsewhere [28], including its performance improvements on load-imbalanced clusters.

The uniform *map* and *reduce* methods, applied by all nodes holding similarly sized data in traditional MapReduce implementations, like Hadoop create performance problems when nodes have *non-uniform processing capabilities*. When some nodes are faster than others they finish their work more quickly. However, the cluster must also wait for the slow nodes to complete their tasks before presenting the output to the user.

In previous work we designed MARLA (MApReduce with adaptive Load balancing for heterogeneous and Load imbalAnced clusters) [28]. MARLA is a MapReduce implementation designed to work directly with existing file systems in a cluster instead of relying on the Hadoop Distributed File System (HDFS) [4]. Whereas Hadoop [3] is closely coupled with HDFS [4], its file system, MARLA [28] and its predecessor MARIANE [27] are built to focus solely upon `map` and `reduce` task management. In order to accomplish this, MARLA depends upon a networked file system, which serves to decouple the data management and the framework itself, allowing the framework and the File System to each perform the task for which they were designed. MARLA was developed specifically for High Performance Scientific Compute clusters, such as those at the National Energy Research Scientific Computing (NERSC) Center [5]. Traditionally, in order for these labs to accommodate Hadoop, the cluster must be partitioned into pieces where some are allocated for use with Hadoop only. In order to eliminate this need, MARLA is able to operate on any existing shared file system such as NFS or GPFS [6]. Incidentally, this also opens up the MapReduce programming model to scientific applications

that require POSIX compliance, which HDFS does not currently support.

Clusters whose nodes possess non-uniform processing capabilities (some nodes faster than others) undermine Hadoop's strategy of partitioning data equally across nodes and applying *map* and *reduce* methods uniformly. Workers on fast nodes finish their work quickly but must wait for straggler workers on slower nodes before the application completes.

MARLA works directly with existing cluster file systems instead of relying on the Hadoop Distributed File System (HDFS) [4]. MARLA and its predecessor MARIANE [27] instead focus solely on `map` and `reduce` task management. MARLA uses a networked file system (e.g. NFS) to decouple data management from the framework, allowing the framework and the file system to address their separate concerns independently. MARLA specifically targets high performance scientific compute clusters, such as those at the National Energy Research Scientific Computing (NERSC) Center [5]. To run Hadoop and HDFS, these HPC centers typically partition their clusters and dedicate sub-parts for exclusive use by Hadoop [27]. MARLA can instead operate on existing shared file systems such as NFS or GPFS [6]. This feature increases the number of nodes available for MapReduce jobs, removes the requirement that individual nodes contain significant local storage, and enables MARLA to support scientific applications that require POSIX compliance.

In MARLA the *Splitter* manages the Input and Output of the framework. This module is responsible for using the framework configuration parameters to divide the input into separate chunks. In this case, the input is split based on two components. The first component is the number of tasks, and the second component

is the number of cores on each of the worker nodes. Both of these values are defined by the user in the framework's configuration file. The number of tasks splits the data into primary input chunks, so that each worker requests a specific task and receives all data corresponding to that task. The number of cores is used to sub-divide a task into smaller chunks, one for each of the specified cores so that the workers can benefit from multi-threaded processing. On the other hand, with Hadoop, the data is split based upon block size where it is divided and replicated across the cluster. The individual data splits are placed based on several factors and placement changes the task assignments of various nodes. This data placement mechanism binds tasks to workers at an early stage. While these bindings are not permanent (tasks can be migrated later) there is a preference implicit in this system, and this preference makes it difficult for Hadoop's *straggler* mitigation technique to adapt when clusters are partially upgraded.

The MARLA *Splitter* manages framework I/O. Framework configuration parameters drive and determine the division of application input into chunks.

Different configuration parameters specify:

- the number of tasks
- the number of cores on each worker node

Workers request tasks and receive all associated input chunk data. To facilitate processing in a heterogeneous environment, MARLA allows the user to configure a number of tasks for the data to be split into. This parameter defines how many data chunks the input should be divided into, which allows the user to adopt a bag-of-tasks approach to combating heterogeneity. After the *Splitter* divides the tasks into input data chunks, it sub-divides those chunks into as many sub-tasks

as there are cores on each worker node, a value defined by a framework parameter. This is done to facilitate multi-threading on worker nodes. When a worker node requests a task, the file handle gets passed as an argument, and the file system ensures that the worker node can access the file.

Hadoop instead splits and replicates data based on block size, and places it based on node storage capacity, among other factors. Data placement influences the nodes on which workers complete tasks, often well before the application runs. Although tasks can migrate from one node to another at the request of the Master, the system's implicit preference toward local tasks makes it difficult for Hadoop's straggler mitigation technique to keep up with the non-uniform processing capability of the cluster nodes when only portions of the cluster have been upgraded as described in the literature[7, 76].

With I/O handled by the *Splitter*, the *TaskController*, also known as the *Master* makes the *map* and *reduce* code provided by the user available to the worker nodes, and starts and stops the MapReduce job. The *TaskController* monitors task progress on behalf of the worker nodes, and resubmits failed tasks to the *FaultTracker*.

The *FaultTracker* monitors which tasks have failed on a task-by-task basis. Nodes that failed to properly complete their assigned task are put on a short temporary leave while their failed task is re-tried. The *FaultTracker* issues a strike to any worker node that failed to complete its assigned task if another worker was able to complete that same task. After three strikes a worker is deemed faulty and blacklisted from further participation in the job.

MARLA's *TaskController*, or *Master*, makes the user's *map* and *reduce* code available to workers, and starts and stops MapReduce jobs. The *TaskController* monitors

task progress on behalf of worker nodes, and resubmits failed tasks to the *FaultTracker*. The *FaultTracker* monitors tasks for failure, issuing a "strike" against any node that fails on a task that a worker on another node successfully completes. Three strikes relegate a worker to a *blacklist*, precluding it from further participation in the job.

Recall that as part of the traditional MapReduce model, a job can only execute as quickly as its slowest task. A slow-running task produces a bottle-neck that can have a negative impact on turn-around time. A task that is slow can be the result of a node that lacks processing power, a node that is burdened with external load, or even variances in input that require additional processing. To be able to adapt to these challenges without making assumptions based upon static performance results, or profiling of nodes, we introduced the bag-of-tasks mechanism to combat both static and dynamic heterogeneity.

Originally, the slowest MapReduce tasks, straggler tasks, limited and determined the turnaround time of larger MapReduce jobs. Causes of straggler tasks include less capable node hardware, external load, and variances in input chunk data, some may require more processing than others. To adapt to these challenges without making assumptions based on static profiling, MARLA supports the bag-of-tasks model to combat both static and dynamic heterogeneity.

With these features in mind, our motivation for this work is to determine the cost associated with the adaptability provided by a bag-of-tasks approach to task management within the context of a MapReduce framework. The work presented in this paper seeks to analyze the optimal configuration required to adapt to a heterogeneous cluster by assigning multiple tasks per node. There is a trade-off

associated with adding more tasks since each task generates additional overhead. At the same time, additional tasks means each individual task is smaller, which allows closer approximations for equivalent workload distribution.

Following this we characterize the performance of this bag-of-tasks approach within a MapReduce framework. We identify beneficial framework configurations for adapting to performance-heterogeneous clusters. Assigning increasing numbers of tasks per node allows frameworks to divide data and tasks to better match node capabilities, but invites overhead.

3.2 Experiments

In order to experimentally simulate small differences in heterogeneity, we perform incremental upgrades of worker nodes within the cluster. As described by Ripal et al. [55], data centers perform incremental upgrades of their compute and storage infrastructures approximately every two years.

Our experiments run on the Binghamton University Grid and Cloud Computing Research Laboratory experimental research cluster, which comprises the following components:

- 1 *Master* node running a 4 core Intel Xeon 5150 @ 2.66GHz and 8 GB RAM
- 24 *Baseline* nodes - 4 core Intel Xeon 5150 @ 2.66GHz and 8 GB RAM
- 24 *Faster* nodes - 8 core Intel Xeon E545 @ 2.33GHz and 8 GB RAM
- 12 *Fastest* nodes - 32 core Intel Xeon E5-2670 @ 2.60GHz and 126 GB RAM

Each node runs 64-bit Linux 2.6.32 and shares an NFS server. To emulate clusters that evolve as described Ripal et al. [55], who report that data centers perform partial upgrades of their compute and storage infrastructures approximately every two years, we model incremental upgrades by enabling different portions of the cluster containing different combinations of the three classes of machines.

We do not include performance data for Hadoop as it does not support deferred binding of tasks. In our earlier work, we compared Hadoop with our MARLA framework for load imbalanced and fault-tolerance scenarios [28]. The comparison shows that MARLA and Hadoop had a similar performance profile for processing floating point data in a homogeneous cluster. However, in 75-node cluster with 600 cores, in which 75% of the nodes have third-party CPU and memory loads, MARLA takes 33% less time than Hadoop to process 300 million matrices. For the widely used MapReduce benchmark of processing a 0.6TB file for word frequency count, Hadoop and MARLA were tested for fault tolerance. In this test, a 32-node cluster progressively lost 6, 8, 10, 12, 14, and 16 nodes. The results showed that MARLA consistently performed better than Hadoop when faced with loss of nodes.

For the experiments in question we multiply matrices containing random floating point values. The CPU-intensity of matrix multiplication emulates the characteristics and requirements of many Big Data applications. The differences between Baseline, Faster, and Fastest nodes lie primarily in processor speeds and the number of cores; therefore, CPU-intensive applications highlight this difference most effectively. We report the following:

- The average time for ten runs of each experiment

- The number of 33×33 matrices that are multiplied

We design and run experiments on a cluster that utilizes a centralized file system (NFS). We limit the scope of this paper to the realm of NFS for two reasons. The first is based on our prior work MARIANE [27], in which we discuss how it is often the case that HPC environments are unable to utilize the MapReduce paradigm because of the burdens imposed by HDFS. The MARLA framework utilizes the same code-base as MARIANE as it was also designed with such HPC environments in mind. A comparison of how the use of HDFS has an effect on the performance of a MapReduce framework in such an environment was previously considered in work out of this lab [27]. The second reason we restrict our experiments to use of a centralized data store is because of evidence that suggests that many companies, like Facebook, use NFS alongside HDFS when processing Big Data [70]. Since HDFS does not support late-binding of tasks to workers, and that is the aspect of this framework we wish to study, we limit our study to an NFS-based environment.

3.2.1 Clusters with Two Levels of Nodes

The first set of experiments varies the cluster configuration, the split granularity (that is, the number of tasks-per-node into which the framework splits the problem), and the input data size. In particular, we run tests for all combinations of the following:

- *Cluster configuration*: 16-node clusters with some Baseline nodes and some Faster nodes, varying the percentages of each in increments of four nodes, or

25% of the cluster nodes.[1]

- *Split granularity*: We vary the number of tasks per node from one to four. To utilize the upgraded nodes most effectively, the number of cores parameter of the MARLA framework is defined as eight. Recall that this parameter defines how many sub-tasks to attribute to each task.

- *Problem size*: We use input matrices of size 33×33 randomly generated floating point values, multiplying 500K, 750K, 1M, 1.25M, 1.5M, 1.75M, 2M, and 2.25M matrices during execution of the various MapReduce jobs.

Section 3.3.1 contains results for this set of experiments.

3.2.2 Clusters with Three Levels of Nodes

The second set of experiments studies the effect of introducing the third class of Fastest nodes. We vary a 24-node cluster to contain all Baseline nodes, and then a variety of upgrade combinations. In particular, we vary the number of Faster nodes from zero to twenty-four, in increments of two. We simultaneously vary the number of Fastest nodes from zero to twelve, in increments two. We use tuple notation $\langle b, f, t \rangle$ to indicate the number of nodes at the $\langle b = Baseline, f = Fast, t = Fastest \rangle$ levels. We run tests for all tuples $\langle b, f, t \rangle$ in the following set: $\{\langle b, f, t \rangle | b \in [0, 24], f \in [0, 24], t \in [0, 12]; 2b, 2f, 2t \in \mathbf{N}; b + f + t = 24\}$.

In this configuration, we also vary the number of cores per worker alongside the number of tasks. This is done to identify what happens when the number of cores in the configuration file is not reflective of the actual number of cores on the

[1] We do not use the Fastest node configuration for this set of experiments.

most powerful of the nodes. To do this we consider splitting the tasks into 8 sub-tasks as we did for the previous experiments; we also consider splitting the tasks into 32 sub-tasks in an effort to take full advantage of the Fastest nodes. As with the previous set of experiments, we also vary the number of tasks. We vary this parameter in the same manner as the previous set of experiments, from one to four times the number of nodes in the cluster.

In an effort to understand the effects of this framework parameter, we add an additional test where we consider a task count equal to five times the number of workers, with a sub-task count set at 32. Section 3.3.6 contains results for this third set of experiments.

3.3 Results

3.3.1 Variable Data Size Through Upgrade

This section describes results from tests that vary three different aspects of a MapReduce matrix multiply application running over MARLA. In particular:

- Increasing the *split granularity*, the number of tasks per worker node into which the original data set is split, provides more opportunity for Faster nodes to receive and complete more work in smaller chunks than slower nodes. In a 16 node cluster, results describe sets of runs with data split into 16 tasks (1 per node), 32 tasks (2 per node), 48 tasks (3 per node), and 64 tasks (4 per node).

- Altering the *performance-heterogeneity* of the cluster influences the degree to

which the system requires straggler mitigation. Results describe sets of runs on a homogeneous system of all Baseline nodes (labeled "0% Faster" in figures), a system with 25% of the system upgraded to Faster nodes, systems with 50% and 75% Faster nodes, and a homogeneous system of 100% Faster nodes.

- Varying the *problem size* ensures that trends exist as computational requirements of the application increase. Experiments set the size of matrices at 33×33 floating point numbers, and set the number of such matrices in the input data at 500K, 750K, 1M, 1.25M, 1.5M, 1.75M, 2M, and 2.25M matrices.

Four split granularities, five performance-heterogeneity levels, and eight input set sizes translate to 160 different tests. Graphs depict the averages of ten runs of each test. We plot portions of the data in several different ways to explore trends and highlight results that provide insight.

3.3.2 Traditional Coarse-Grained Splits

Figure 3.1 plots only the data for the most coarse grain split granularity of one task per worker node. This split mirrors the default behavior in Hadoop and explicitly disallows straggler mitigation because all nodes (no matter their capability) receive exactly one task at the outset of the application. Each group of five bars corresponds to a different problem size along the x-axis, the y-axis reflects execution time, and each bar corresponds to a different performance-heterogeneity (or upgrade level). Larger problem sizes take longer to finish, and clusters with 75% and 100% upgraded nodes outperform less capable clusters. However, a homogeneous cluster with all Baseline nodes, and clusters with 25% and 50% upgraded

nodes all perform the same.

To understand this behavior, consider an example. Suppose we have N worker nodes and we assign $N + 1$ approximately equal sized tasks to each of them. In order for this running time to be comparable to the case where we have N tasks for N nodes, we would need a cluster configured in such a way that the fastest node is nearly twice as fast as the slowest node. In this scenario, the fastest node takes two tasks of equal size, and the slowest node takes one task of that same size. This implies that the execution time of the job is not related simply to the speed of the slowest node, but to the speed of the fastest node relative to the slowest node.

Expanding this example shows us that in order for our cluster to be able to achieve a performance improvement with $3N$ tasks per worker, the fastest node would have to be able to compute at least one of the slowest node's tasks; meaning that the fastest node would have to complete three tasks before the slowest node could finish two tasks. Note that the turnaround time in this case will then depend on the ability of the fastest node to complete four tasks, but that it is sufficient to complete three tasks before the slowest node completes two tasks. This is because once three tasks have been completed by the fastest node it will be free to request, and receive, more work from the Master which will prevent the slowest node from receiving that same work. In this scenario, the fastest worker node would have to be just over 1.5 times the speed of the slowest worker. In addition to this, there would need to be enough faster nodes in the cluster to be able to prevent all of the slower nodes from requesting an additional (third) task.

Because of this, for a traditional coarse-grain data split, initial upgrades to the cluster (even upgrading half of the cluster to machines with faster processors and

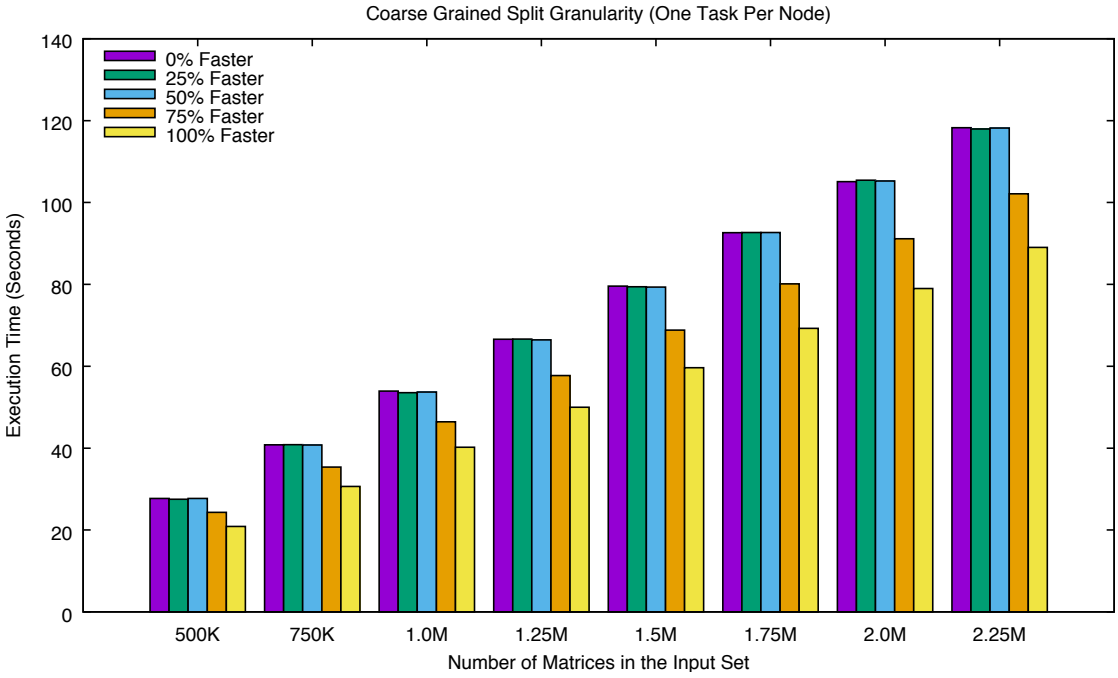

Figure 3.1: Execution time for the traditional one-task-per-worker initial data split, for different problem sizes and cluster upgrade levels

twice as many cores) does not improve matrix multiplication performance. Overall application performance depends on stragglers on slower nodes, and the coarse grain split precludes straggler mitigation. Upgrading most (75%) or all of the cluster to Faster nodes does improve performance.

3.3.3 Progressive Granularity Changes

In order to analyze what happens as we move from coarse granularity to fine granularity with respect to the number of tasks, we perform experiments for each multiple of the number of worker nodes as we move from one task per node to four tasks per node. The results of this class of experiments follows are similar to those

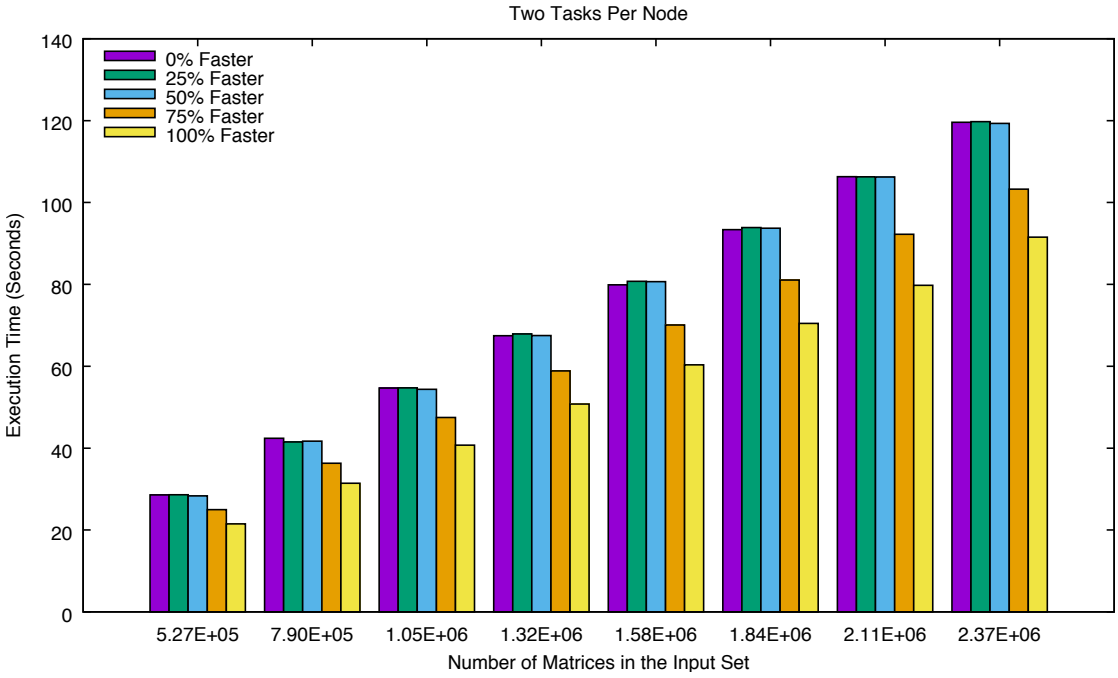

Figure 3.2: The execution times of workloads when we follow a two tasks per worker splitting rule as we incrementally perform upgrades on a subset of the nodes.

seen in Figure 3.1 and the analysis provided in Section 3.3.2.

Doubling The Number of Tasks

In this set of experiments, we consider what happens when we double the number of tasks from one task per worker to two. These results are presented in Figure 3.2. When we compare these results with those found in Figure 3.1 and described in Section 3.3.2, they are very similar. The largest degree of difference between the two sets of experiments is 3.77%. The average degree of difference between a one task per worker setup and the two task per worker setup is only 1.72%. The increase in execution time is as a result of the overhead associated with worker nodes having to request additional work.

To illustrate the effects of the overhead, we consider the change in execution time between one task per worker and two tasks per worker considering file size. The difference is 2.91% on average for the smallest file size tested, but this difference steadily decreases as low as 1.01% as the file sizes increase. Since this difference is not as prominent for the larger file sizes, we determine that the overhead associated with requesting additional work is relatively constant and does not depend on problem size. This tells us that the percentage overhead associated becomes amortized. From this we can conclude that as long as the file size is reasonably large, the cost of adding more tasks from the same data will not produce a heavy negative impact on our execution times provided the adaptability to heterogeneity is necessary.

Further, we expect similar performance for the one task per node and two tasks per node schemas in most cases. The reason that we expect this is because two tasks per node does not allow much room for adaptability to heterogeneity. To illustrate the inability to adapt at this level of tasks granularity, we present Figure 3.3. In the figure, the execution time of workloads is presented relative to their execution time on the original (unupgraded) cluster. The trend of the data presented is that the execution times for the slightly upgraded clusters (25% and 50% upgraded) are relatively consistent. These results also indicate a slight increase in execution time for smaller file sizes relative to larger file sizes. This illustrates both the overhead and the inability to adapt to heterogeneity at this level of task granularity. In order to realize heterogeneity adaptability from such a small difference in the number of tasks, the degree of heterogeneity would have to be high. In particular, for a cluster to be able to utilize the creation of one additional task per worker,

a fast node would have to be able to reliably complete two tasks before the slowest node finished one task. Otherwise, the slowest node would be able to request its second task and the entire job would complete only once the slowest node completes its second task. In short, an upgraded node would have to be roughly twice as fast as an non-upgraded node in order to see an improvement in execution time when there are two tasks per node.

This has been partially addressed by the Tarazu Enhancement Suite [7] when considering clusters that have both extremely fast and extremely slow nodes. In particular, they consider a heterogeneous cluster of Intel Xeon server class hardware and Intel Atom hardware. They discover that in Hadoop, "work stealing" occurs, and is especially a problem toward the end of the `map` phase. In this scenario, the Xeon nodes speculatively execute tasks whose data are local to the Atom nodes. The non-local status of the data associated with the task is the cause of this "work stealing" happening at the end of the `map` phase, since Hadoop's speculative execution prefers local tasks to remote ones. As a result of "work stealing", the Xeon nodes prevent the Atom nodes from preforming their fair share of work. Tarazu [7], a Hadoop enhancement suite mitigated this problem by adding in communication aware speculative execution of tasks. However, this late `map` phase "work stealing" is not a problem in our framework. Due to the data visibility afforded to MARLA by use of a networked file system, the concept of remote tasks does not exist. Additionally, MARLA has no preference for local tasks over remote tasks, as all data is visible to all nodes. The Tarazu enhanced Hadoop has not been analyzed within the context of upgrading clusters as we have presented here; it focuses on the disparity between computation capabilities of Xeon and Atom based

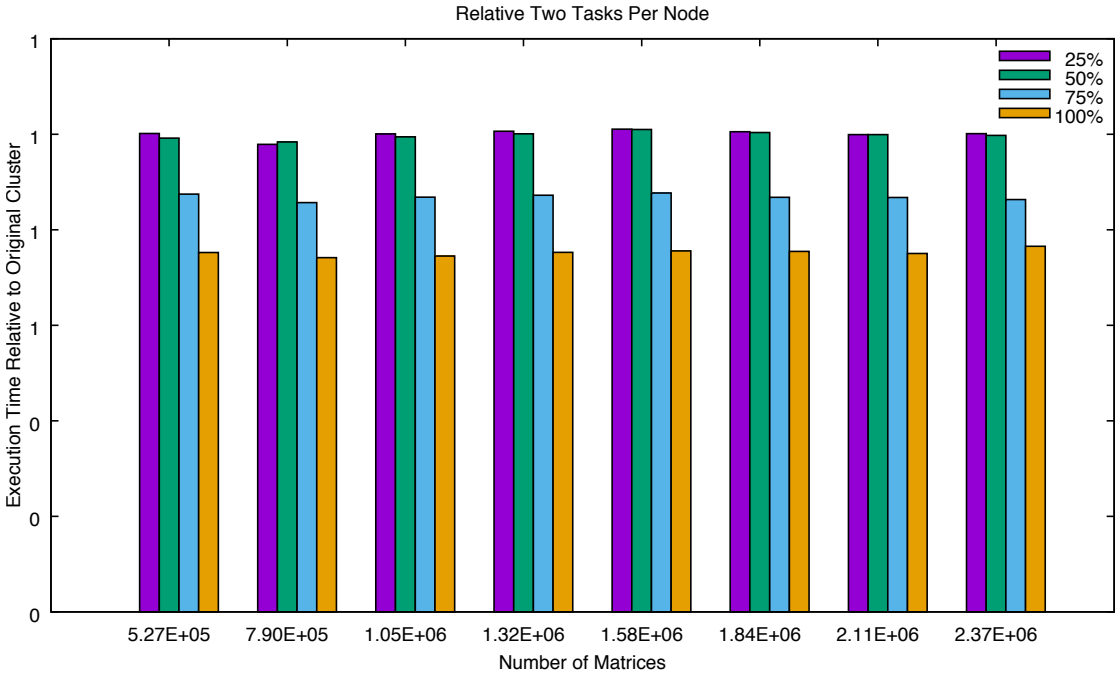

Figure 3.3: Overhead for when we follow a two tasks per worker splitting rule as we incrementally perform upgrades on a subset of the nodes.

hardware, a drastic difference. We believe that a single node being upgraded to a node whose hardware is slightly more powerful is a more plausible scenario for existing data centers, as opposed to replacing one *brawny* node with eight *wimpy* ones.

Further Splitting Tasks The results presented here correspond to what happens when the number of tasks available is three times the number of worker nodes. We look at these results explicitly, as well as comparing them to the results obtained in previous sections.

As with the previous granularity shift, we expect some deviation from the previous results because three tasks per node allows slightly more room to adapt to heterogeneity. From our results, the amount of overhead increases as we increase

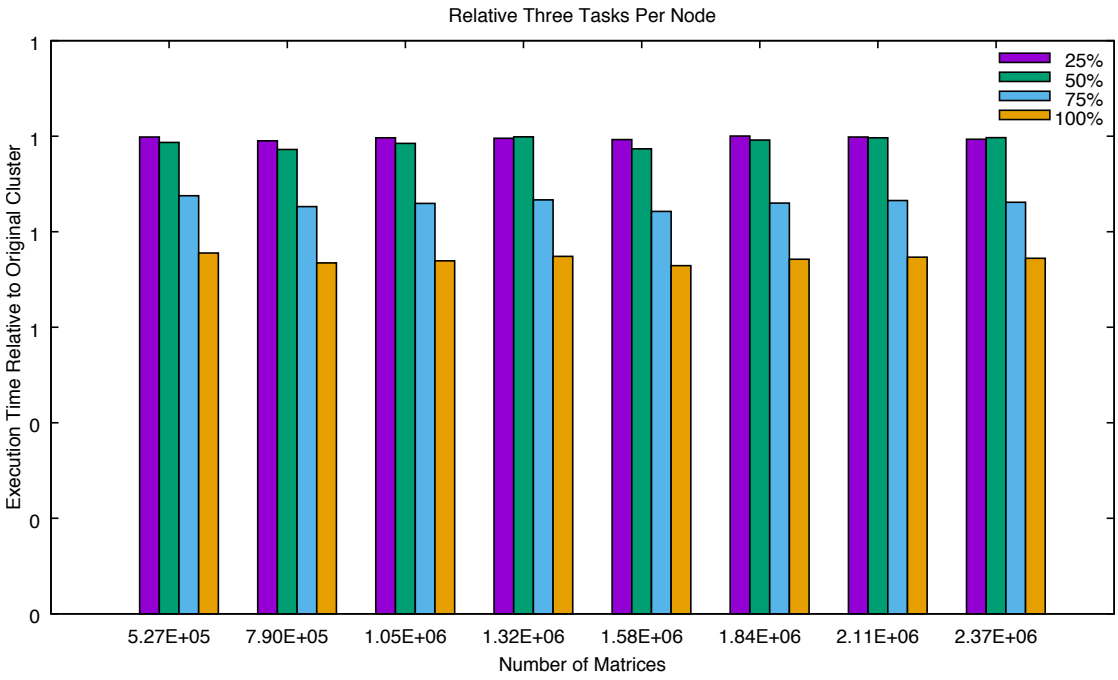

Figure 3.4: Overhead for when we follow a three tasks per worker splitting rule as we incrementally perform upgrades on a subset of the nodes.

the number of tasks. This is expected because additional return trips to the master node are required to be assigned more work. This introduces stalling between tasks, which reduces turn around time. In particular when these results are compared to those presented in the previous section, the additional overhead as a result of generating yet more tasks can be seen. In this case we see a maximum overhead of 4.90% and an average overhead of 1.88%. Additionally, it is again confirmed that overhead is amortized as the problem size grows, ranging from 3.19% overhead as we move from two to three tasks per worker for the smallest problem size to 0.48% overhead for the largest problem size.

Next consider the results seen in Figure 3.4. Here we expect to see similar results to that of 3.3. The data presented in this figure displays a slight change between a 25% upgraded cluster and a 50% upgraded cluster, a trend which is not seen in Figure 3.3. This shows that while still not entirely able to adapt to this particular level of performance-heterogeneity, three tasks per worker shows slight performance improvements as smaller sections of the cluster are upgraded. This indicates that further increasing the number of tasks will likely have a more dramatic impact on turnaround time for these smaller percentage upgrade scenarios. This is something we consider in the next section.

Note that in the data presented thus far our cluster is not heterogeneous enough, nor the task granularity small enough, to see an improvement in performance using the configurations presented. The reason we have not yet seen performance improvements in most upgrade scenarios tested is because the upgraded nodes are not fast enough as to be able to take over the execution of all additional tasks that would be assigned to the stock (original) nodes when assuming all nodes will

process the same number of tasks.

3.3.4 Finer-Grained Splits

Figure 3.5 plots data for the same set of tests as Figure 3.1, for the finest granularity of the initial data split. This provides the most potential for Faster nodes to complete initial small assignments quickly and then retrieve more data and execute more tasks than slower nodes. In this case the Baseline homogeneous cluster (0% Faster nodes) performs worst across all problem sizes, and the Faster homogeneous (100% Faster) cluster performs best, two unsurprising results. The other three clusters, however, perform very similarly to one another across all problem sizes, despite the disparity between the number of upgraded nodes.

For a finer-grain data split, MARLA[28] improves performance when the first 25% of the cluster is upgraded, but subsequent upgrades to 50% and 75% do not yield performance gains. Only when the entire cluster runs Faster nodes do we see the next level of application performance improvement.

We plotted but did not include results for the 32-tasks (2 per node) and 48-tasks (3 per node) versions of Figures 3.1 and 3.5. The two omitted graphs plot data whose values closely approximates the data for Figure 3.1. For example, runtimes are only slightly longer in all cases for the 2 tasks-per-node experiments (on average, values are 1.75% longer and each individual value is within 4% of its counterpart). The small increase reflects the small overhead of worker nodes requesting extra work rather than receiving it in the initial split. Figure 3.6, described next, adequately demonstrates the similarity of the data for the omitted graphs.

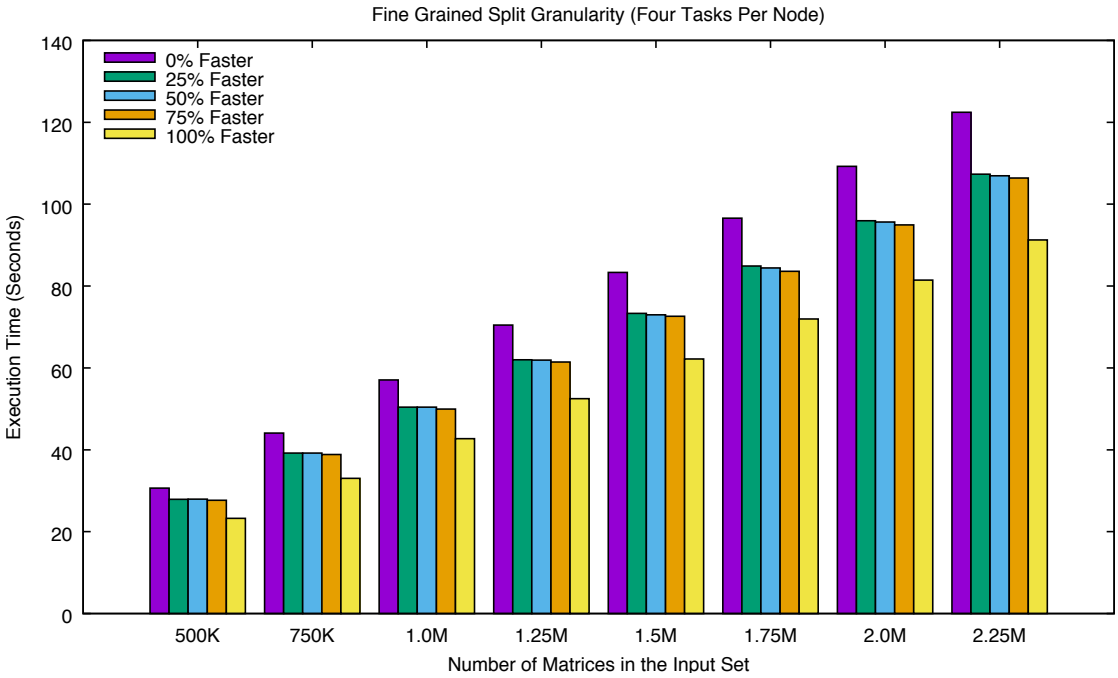

Figure 3.5: The execution times of workloads when we follow a four tasks per worker splitting rule as we incrementally perform upgrades on a subset of the nodes.

3.3.5 Matrices Per Second

As problem sizes grow linearly (e.g. the x-axis in Figures 3.1 and 3.5) the average runtime for each set of tests also grows linearly. To view more data in one place, and to highlight the effect of performance-heterogeneity and split granularity across all tests, Figure 3.6 plots the average number of matrices multiplied per second, for all six cluster configurations paired with all four split granularities. Bars corresponding to the Baseline homogeneous (0% Faster) and the 25% and 50% upgrades reflect similar runtimes for splits of one, two, and three tasks per worker, with a small decrease in slope reflecting overhead. Similarly, all four splits perform better at 75% Faster and Homogeneous (100%) Faster upgrade levels, across all task split granularities, including 4 tasks per node. With the finest grain split that we tested, performance on the intermediate (25% and 50% upgrade levels) clusters closely matches performance on the 75% upgrade level.

Thus, an application developer using a traditional coarse-grain split into 1 task per node would not benefit from any incremental upgrades of subsets of cluster nodes. Only when the entire cluster contains Faster nodes does performance increase. Splitting an application into too few tasks (even 2 or 3 per worker node) similarly does not allow the application to benefit from partial upgrades. Only when the application splits into 4 tasks per worker does the application developer benefit from incremental cluster upgrades. Even then, only an upgrade of the first and last 25% of nodes improves performance. The upgrade of the first 25% allows straggler mitigation strategies to become effective, and the upgrade of the last 25% helps reduce the appearance of stragglers by turning the cluster homogeneous.

We plot Figure 3.6's data differently in Figure 3.7. The downward trend across

results within each of the four set of bars depicts the overhead associated with worker nodes having to retrieve more work, rather than receiving one initial task. In the two homogeneous clusters, the leftmost and rightmost sets of bars in Figure 3.7, this overhead does not pay dividends for any split granularities; the trend continues through all four bars in both cases. Likewise, it does not pay dividends for a cluster with 75% of its nodes upgraded. Within the 25% and 50% upgrade levels, only the rightmost bar is taller, illustrating the need for enough (4) tasks per node in the split to realize improved performance from the first 25% of nodes being upgraded.

Conclusions: Cluster managers should not necessarily expect application performance to improve at all due to partial upgrades, especially when the MapReduce framework employs a traditional one-task-per-worker data split. Our results in as displayed in Figures 3.6 and 3.7 suggest that even MapReduce frameworks that attempt to mitigate the effect of stragglers through the creation of additional tasks may succeed only in adding overhead, and not decreasing runtimes when they do not provide an adequate number of additional tasks. Such frameworks *can*, however, reap the benefits of partial upgrades with sufficient split granularities. In our tests, upgrading the first 25% of nodes allowed MARLA to mitigate the effect of stragglers well enough to have matrix multiply perform as well as a more capable cluster that included 75% Faster nodes.

3.3.6 Variability Between Upgrades

In this set of experiments, we don't simply vary the nodes of the cluster over one upgrade; instead, we consider the possibility of multiple upgrade options.

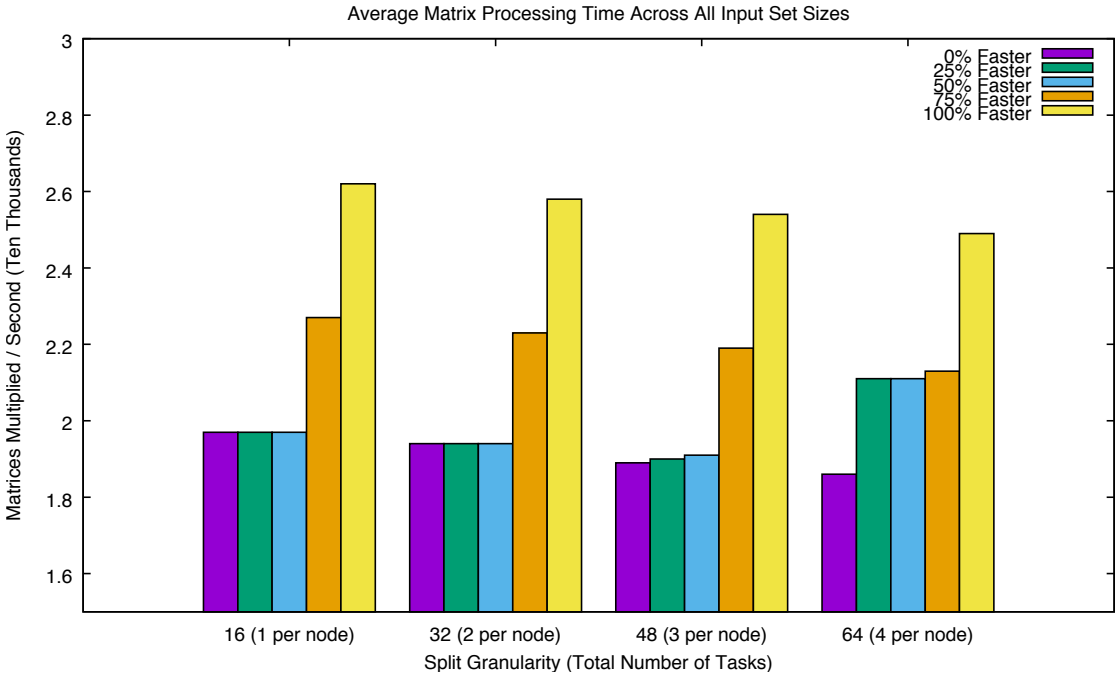

Figure 3.6: Average number of matrices processed per second for different task per node ratios; results averaged across all eight problem sizes.

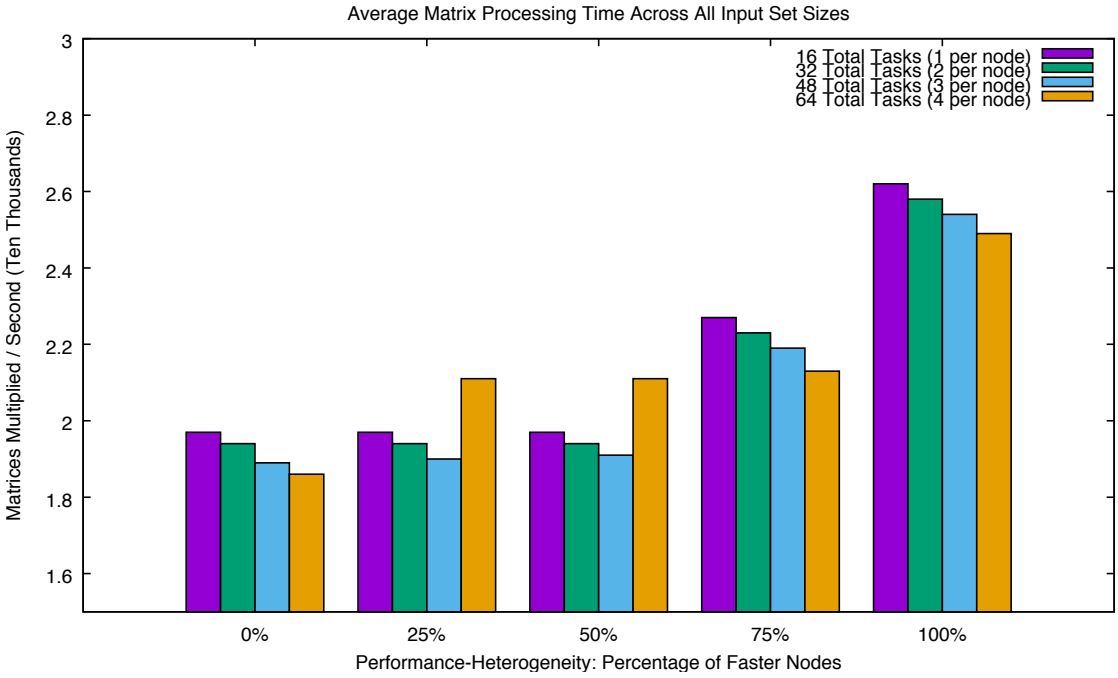

Figure 3.7: Average number of matrices processed per second for different cluster upgrades; results averaged across all eight problem sizes.

When dividing the data for this set of experiments, the data was divided into 24 tasks (one for each node of the cluster), and each of those tasks was sub-divided into 8 sub-tasks (one for each of the cores in the Faster nodes). This was done to see how adaptable the MARLA framework is to a misconfiguration.

The results of this experimentation are presented in Figure 3.8 and illustrate that there are two configurations of the cluster that produce approximately the same run-time. These configurations occur as follows:

- $\langle(65, 1.075), (16, 8.010)\rangle$
- $\langle(65, 1.075), (32, 8.010)\rangle$

In this case, the speedup experienced by the 8-core nodes relative to the 4-core nodes is 1.075; the corresponding speedup for 32-core nodes is 8.010. The runtime of the application when the cluster is in either of these configurations is approximately the same as when the cluster was completely upgraded once. This tells us that even though a minimal number of tasks have been generated and the cluster is not fully utilizing the *fastest* nodes, we can see improved run times.

We introduce the following notation to facilitate discussion of this section's experiments and results. A series of tuples, $\langle(p_i, s_i)\rangle$, describes the heterogeneity of a cluster configuration, where p_i represents the percentage of the cluster that has a speedup of s_i over the slowest node configuration. As a result of this notation, we can accurately express the heterogeneity of the cluster with respect to N classes of hardware, each represented by one entry in a vector of size N. In this vector, the sum of all p_i values is 100. When MARLA configuration causes tasks to sub-divide into eight subtasks at each node, we observe speedup on Faster nodes to be 1.075, and speedup on Fastest nodes to be 8.010. These numbers reflect performance on

homogeneous clusters of Baseline, Faster, and Fastest nodes when running matrix multiplication.

This section describes results of experiments on a cluster that includes a third class of compute nodes, namely the 32-core Fastest nodes described in Section 3.2. Figure 3.8 shows results for an initial split of the matrix multiplication application into 24 tasks (one per node), and for MARLA configured to split tasks into eight subtasks at each node. Therefore, even the 32 node cluster uses 8 cores at a time for each task. When we consider these results, we see two regions that produce optimal run-times, namely:

- $\langle(19, 1.0), (65, 1.075), (16, 8.010)\rangle$
- $\langle(3, 1.0), (65, 1.075), (32, 8.010)\rangle$

Application runtimes for both of these configurations approximate those for the Faster homogeneous cluster configuration, which appears in the lower right corner of Figure 3.8, with 100% Faster nodes and 0% Fastest nodes. Even for a coarse grained task split of one task per node on a cluster configuration that does not take full advantage of the Fastest nodes, run times improve.

Similarly to the previous set of experiments, we consider what happens when we increase the number of tasks, but still assume that each worker has a total of 8 cores. These results, presented in Figure 3.9 shows several trends. The first trend illustrates that despite additional upgrades it is possible for performance to decrease. This occurs in the configurations:

- $\langle(32, 1.075), (16, 8.010)\rangle$
- $\langle(64, 1.075), (16, 8.010)\rangle$

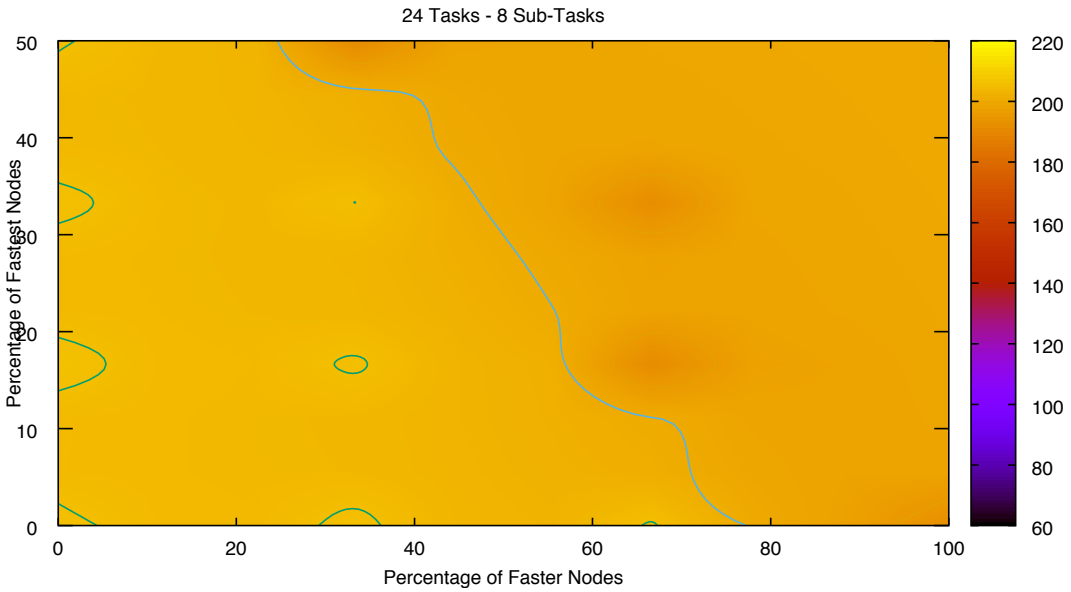

Figure 3.8: Contour plot showing the effects on computation time of upgrading nodes in a cluster with 24 tasks in a 24 node cluster, assuming 8 sub-tasks per task.

This occurs because there are tasks that are pulled by the Faster nodes when they request additional work that cannot be complete. In other configurations, the tasks are able to be completed by the Fastest nodes instead. Another trend we see comes as a result of comparing Figures 3.8 and 3.9. We can see that with 72 tasks instead of 24, the framework is able to adapt to the upgrades provided to the cluster more efficiently. After a smaller percentage of the cluster has been upgraded we are able to see benefits in execution time.

Figure 3.9 shows results for an initial task split of 72 tasks, or three per worker. Again, MARLA splits tasks into 8 subtasks at each node. Figure 3.9 shows that some upgrades result in performance degradation. In particular, the following configurations under perform surrounding data points:

- $\langle(52, 1.0), (32, 1.075), (16, 8.010)\rangle$
- $\langle(20, 1.0), (64, 1.075), (16, 8.010)\rangle$

In this case, Faster nodes request additional work that they cannot complete to improve turnaround time, because requests arrive after the new Fastest nodes have started executing additional tasks. The new tasks on Faster nodes then increase the turnaround time as the framework waits for them to finish. In other configurations, the Fastest nodes can complete these tasks because they constitute a higher percentage of the cluster and are able to get to these tasks before the Faster nodes can.

Comparing Figures 3.8 and 3.9 shows that a split granularity of 72 tasks instead of 24 enables MARLA to adapt to cluster upgrades more efficiently. The difference in performance between these two figures illustrates that with a finer task granularity, upgrades to fewer nodes can still lead to faster execution times.

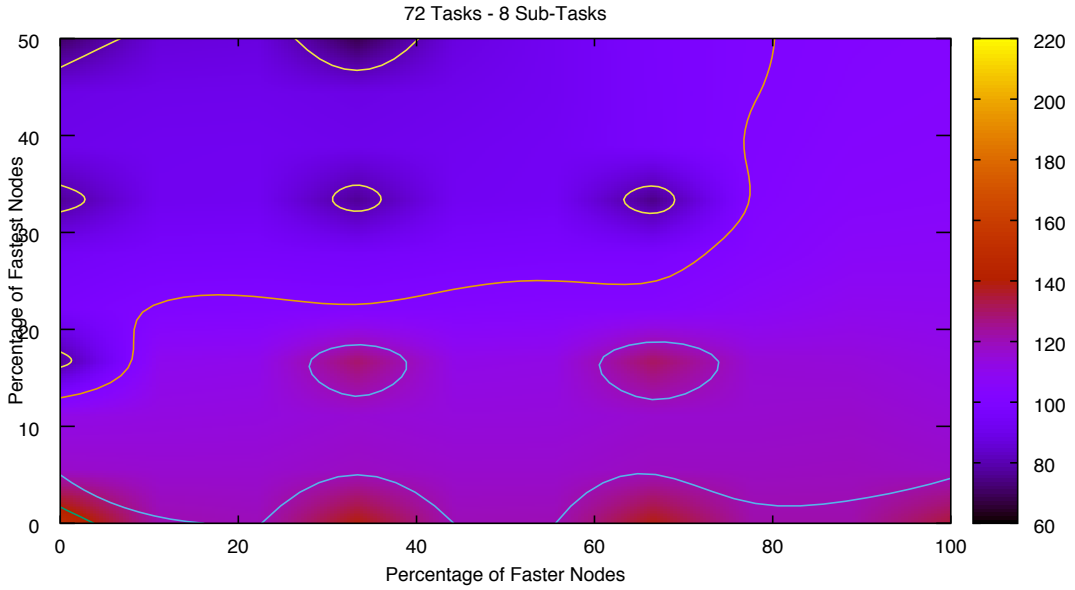

Figure 3.9: A contour plot that shows the effects on execution time upgrading nodes within a cluster with 72 tasks in a 24 node cluster and 8 sub-tasks for each task.

Following these experiments, we conducted experiments that assumed that after the upgrades to the cluster had been made MARLA had been re-configured to assume that each worker consists of 32-cores. The results presented in Figure 3.10 indicate an unanticipated effect of the late binding of tasks to workers. Since tasks are bound to workers after the data has been split, MARLA depends on a cluster topology, depending only on two parameters, the number of tasks and the number of cores per node to split the data. This figure shows that when there are not enough tasks generated, the cluster is slowed down by slower nodes attempting to process more subtasks than it has cores. This can be seen by examining the time difference at configurations $\langle(65, 1.075), (16, 8.010)\rangle$ and $\langle(65, 1.075), (32, 8.010)\rangle$ relative to those times in Figure 3.8. The best performance is achieved in a larger range of configurations when each node is processing only 8 sub-tasks as compared to when each node is processing 32 sub-tasks. This provides for two conclusions:

- The one-task per worker idiom is particularly ineffective when a MapReduce framework is not correctly configured to match the current cluster topology
- MARLA would benefit from a mechanism that adapts to a cluster's topology without having to sacrifice the late binding of tasks to workers

We also consider configurations where MARLA divides tasks into 32 sub-tasks. Figure 3.10 indicate that when too few tasks exist, Baseline nodes incur the overhead of 32 subtasks on a 4 core machine. This effect appears in the time difference at configurations $\langle(19, 1.0), (65, 1.075), (16, 8.010)\rangle$ and $\langle(3, 1.0), (65, 1.075), (32, 8.010)\rangle$ relative to the corresponding points in Figure 3.8. The best performance is achieved in a larger range of configurations when each node processes 8 sub-tasks instead of

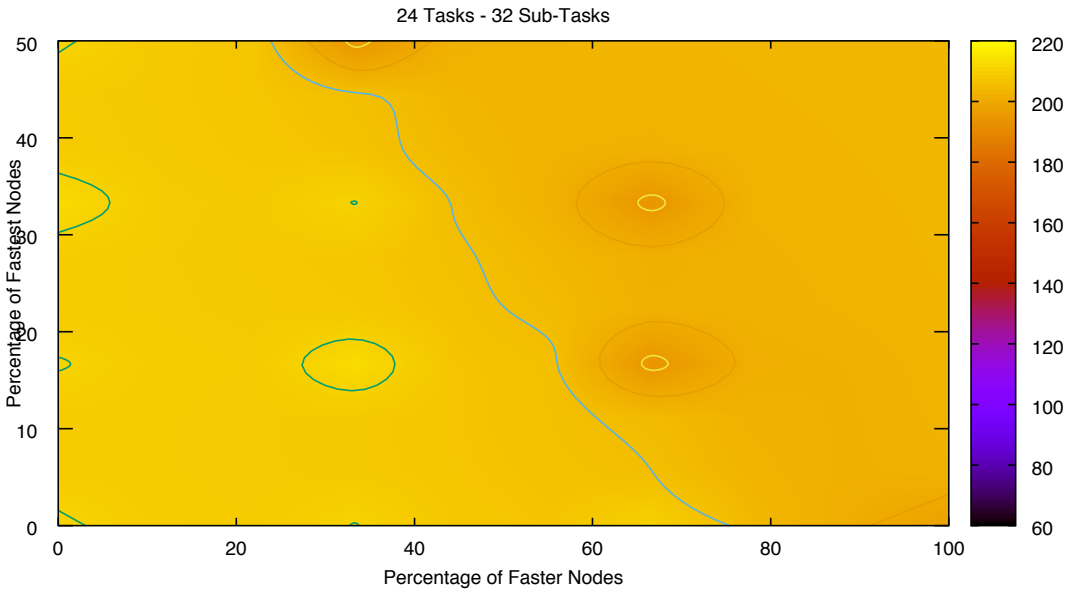

Figure 3.10: A contour plot that shows the effects on computation time of upgrading nodes within a cluster given 24 tasks in a 24 node cluster and 32 sub-tasks per task.

32. Therefore, the one-task per worker heuristic fails when a MapReduce configuration does not match the cluster topology. Furthermore, MARLA could improve by adapting to a cluster's topology without sacrificing the late binding of tasks to workers; we plan to study this as future work.

When we consider however, that MARLA was designed to break the one task per worker idiom that is typical of MapReduce frameworks, it is necessary to consider what happens when we employ the bag-of-tasks technique that gives MARLA its ability to adapt to performance-heterogeneous clusters. Consider the data in Figure 3.11, we can see that upgrading has more of an impact on application turn-around time when the task granularity is smaller and the sub-tasks are

defined appropriately. In this configuration, with 72 tasks and 32 sub-tasks per task, we can see that after 12.5% of the cluster has been upgraded twice our execution time drops below 100 seconds. Additionally, we can see that configurations that produced worse performance when we had only 8 sub-tasks per task, as seen in Figure 3.9, produced better performance with this setup. This is because in the previous setup, when a Fastest node acquired a task, it was only able to run it on 8 of the 32 available cores and now it is able to fully utilize all 32 available cores. In this case, performance is more effectively improved by the smaller task granularity when the most powerful nodes in the cluster can be effectively utilized.

Figure 3.11 shows results of dividing work onto 72 tasks (three per worker), and shows that upgrading impacts application turn-around time for smaller task granularities and for systems whose MARLA number-of-cores parameter is set properly. With 72 tasks and 32 sub-tasks per task, and more than 12.5% Fastest nodes, execution time drops below 100 seconds. Further, configurations that split into 32 nodes perform better than when tasks split into only 8 subtasks, because the Fastest nodes can use all 32 cores. In this case, performance improves more effectively when the most powerful nodes in the cluster are using effectively utilized.

This section's results indicate:

- The one task per worker heuristic combats *performance-heterogeneity* for the configurations we tested, with three discrete levels of worker performance.
- A misconfiguration of a MapReduce framework that is not fully aware of cluster topology can reduce the number of configurations that provide improved performance as the cluster is upgraded.

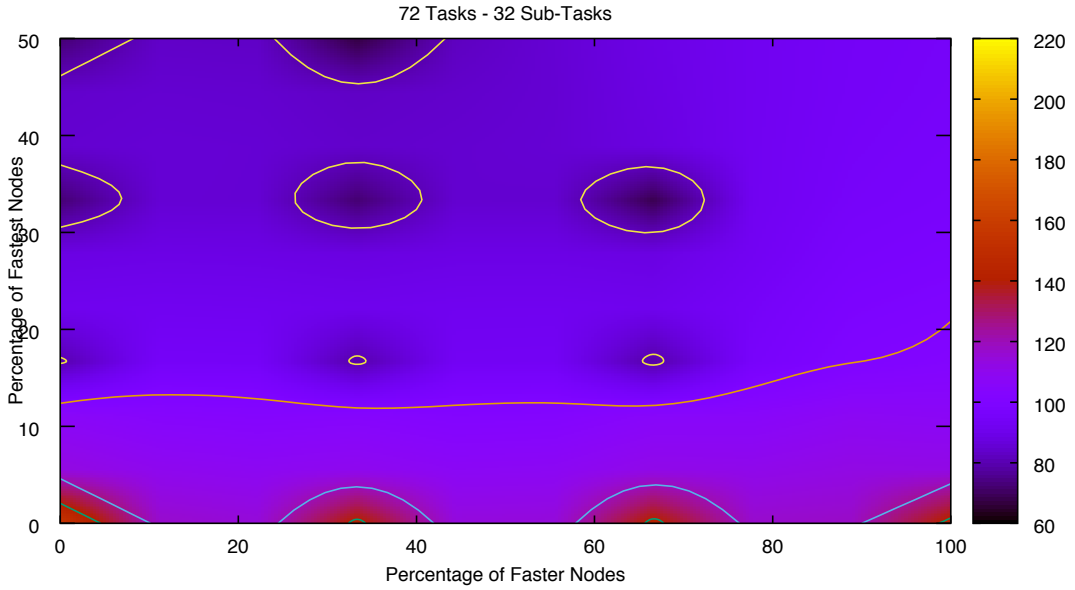

Figure 3.11: A contour plot that shows the effects on computation time of upgrading nodes in a cluster given 72 tasks in a 24 node cluster and 32 sub-tasks for each task.

- The addition of more tasks to the pool of tasks that needs to be completed allows for a MapReduce framework to be configured so that it takes full advantage of the Fastest nodes in the cluster and still sees improved turn-around time for most cluster configurations.

3.4 Conclusion

3.4.1 Conclusions

As we discussed in MARLA [28], we are able to accommodate heterogeneity in a cluster by increasing the number of tasks associated with each worker node. Thus far using experimentation on variable data sizes, variable degrees of heterogeneity in the cluster, and various data partitioning rules we are able to provide the following results:

- As the processing data size grows 4.5 fold, the amount of overhead produced as a result of an increased number of tasks decreases, resulting in performance improvement only when the file size is large. In the case of a four-task-per worker ratio, the overall execution time increases by an average of 7.553% in the case of the smallest file, and decreases by an average of 1.661% in the case of the largest file. Therefore, frameworks should consider heterogeneity mitigation using a bag-of-tasks mechanism only when the file size is large.

- An increase in task granularity can provide performance improvements even

in clusters that do not have a high degree of heterogeneity. For example, increasing task granularity from two tasks per worker to four tasks per worker generates, on average a 3.13% improvement in execution time across our runs executed using the largest input file. In particular, improvements are seen in as little as a 25% cluster upgrade in the case of four tasks per node; whereas improvements are not seen until a 75% upgrade for the two tasks per node case.

- Higher task to worker ratios increase performance more for clusters that have a small percentage of fast nodes than those with a small percentage of slow nodes. In fact, for the largest data and a 75% upgraded cluster, increasing the task to worker ratio from two to four caused a 3.03% execution time increase. This is due to the overhead associated with the additional tasks. Whereas, for a 25% upgraded cluster a 10.36% decrease in execution time was seen.

- The degree of heterogeneity is not only a factor of how many nodes are different, but also the difference in computing ability of the various types of nodes. This degree of heterogeneity can be used to help determine the optimal number of tasks that should be used to mitigate performance-heterogeneity in a cluster.

The conclusions above are illustrated in Figure 3.12. This figure displays the average execution time per task, normalized based upon the data size. As we increase the number of tasks, we can see that performance decreases when the cluster is homogeneous due to the additional overhead associated with these tasks. Performance improvements are not seen even though there are upgrades to the cluster,

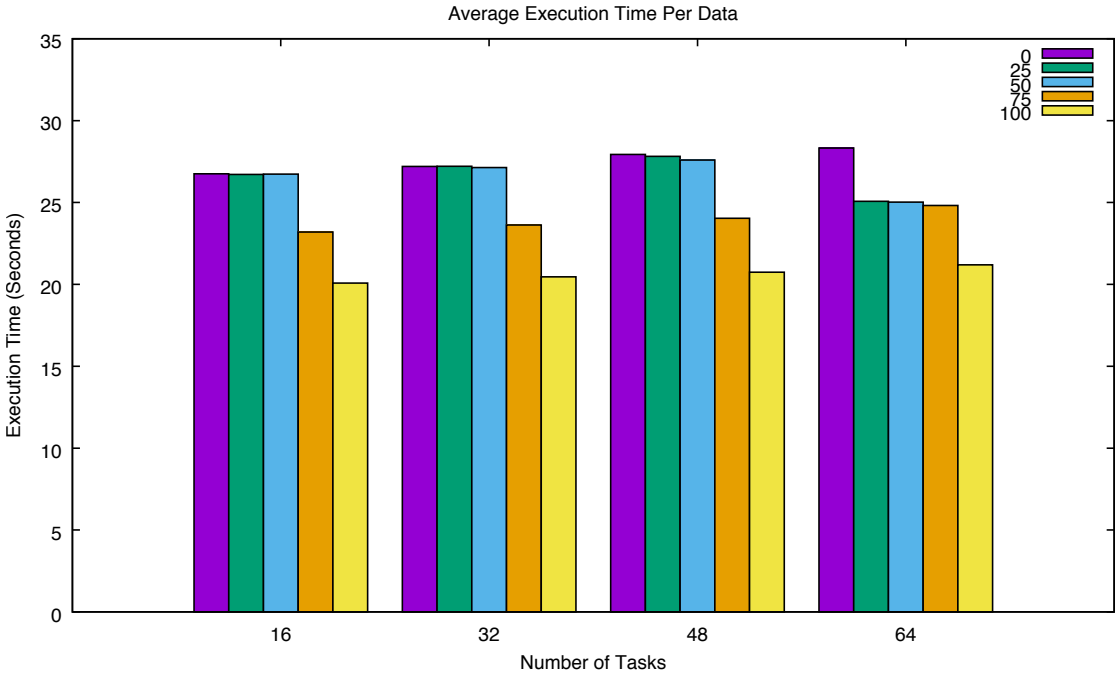

Figure 3.12: Comparison as the number of tasks per worker increases while upgrades are performed on subsets of the cluster, normalized based on data size

as in the two and three tasks per worker cases, since the degree of heterogeneity physically provided by these upgrades is small. Despite the additional overhead associated with generation of four tasks per node, we can see performance improvements based upon the degree of heterogeneity within the cluster.

The degree of performance-heterogeneity in a cluster influences MapReduce application performance. Some MapReduce frameworks can use relatively few upgraded nodes for straggler mitigation and improved performance. But not all upgrades influence performance equally. For example, applications that may benefit significantly from upgrades to the first 25% of nodes, may see no further improvements in upgrades of an additional 25% and even 50% of nodes. MARLA's fine grained splitting of jobs into a larger number of smaller tasks, and further

splitting each task into one sub-task per core (on the cluster node with the most cores) yields the best results for clusters with the most performance-heterogeneity. For homogeneous clusters, however, having many tasks and sub-tasks introduces overhead to tackle a straggler problem that is less pronounced. Clusters with as few as three different classes of nodes can exhibit particular configurations that support significantly improved performance, but not every upgrade automatically leads to requisite performance gains.

Referring to the results of our experimentation as presented in Sections 3.3.1 and 3.3.6, we are able to arrive at several guidelines for configuring a MapReduce framework to be able to handle heterogeneity.

- We should not try to accommodate heterogeneity if it is not to be expected. Namely, if a cluster is dedicated and does not have a large degree of heterogeneity within the nodes, we should not vary from the traditional one task per worker model using the work sharing algorithm that MARLA employs. This can be seen through our analysis provided in Section 3.3.1.

- If a significant portion of the cluster is faster than the remainder of the cluster, the performance of the job while running on the cluster will be approximately the same as if the cluster were homogeneous. We believe that this percentage is directly related to the degree of processing performance difference between different types of nodes.

- There are several factors that affect the ability of a MapReduce framework to respond to heterogeneity by increasing the number of tasks per worker node. More specifically, when dividing input into tasks to be completed by worker

nodes, we see a converging wave pattern that depends on several factors as summarized below.

- The difference in processing power between the fastest and slowest nodes in the cluster determines whether or not a MapReduce job will perform better as more tasks are added to the bag-of-tasks. This means that the amount of slowdown and speedup that we see depends heavily on the difference in processing power between the various kinds of nodes. This variance also determines what percentage of the cluster must be upgraded in order to see a benefit from the bag-of-tasks approach to heterogeneity management. We present a new notation to describe the degree of heterogeneity within a cluster as a vector of N tuples of the form $\langle(p_i, s_i)\rangle$ where p_i is the percentage of the cluster with speedup s_i over the slowest nodes, and the sum of all p_i from 1 to N is 100.

- The size of the data being processed dictates the relative overhead associated with introduction of a bag-of-tasks. Our performance data shows that in order to effectively configure a MapReduce cluster for optimal processing in heterogeneous conditions, we must have sufficiently large data. As the data size increases, we are more likely to require additional tasks to adapt to heterogeneity since the execution time of the data would be longer per task.

- The wave pattern indicates that it is better to assign a number of tasks that approaches multiples of the number of nodes in the cloud from the left. This optimal configuration provides slow nodes fewer opportunities to take additional tasks.

From our results, we can also see that in many cases the execution time of a job remains approximately the same until a certain degree of heterogeneity is attained. Therefore the amount of heterogeneity $\langle (p, s) \rangle$ in the cluster helps to determine the ideal cluster configuration needed for a MapReduce framework to benefit from cluster upgrades without reconfiguration.

As a result of this work it can be seen that upgrades to clusters that run MapReduce frameworks for Big Data processing should be considered carefully – taking into consideration the MapReduce framework's method for correcting the effects of heterogeneity, the degree of heterogeneity introduced, and the types of applications that the MapReduce framework frequently runs. We show that without considering some of these factors, the window for optimal performance can decrease with upgrades to the cluster, rater than increase.

Chapter 4

Dynamic Heterogeneity

Author's note: The text in this chapter is largely from my journal paper "Performance Analysis of Adapting a MapReduce Framework to Dynamically Accommodate Heterogeneity" [33]

4.1 Addressing Inefficient Cluster Usage

MARLA [28] is designed such that it takes the number of cores of a worker as a parameter to the framework. However, this solution has several drawbacks. As shown in our prior work [32] this can lead to inefficient use of the cluster when this parameter is mis-configured. In order to remedy this, a Dynamic Heterogeneity Awareness (DHA) Module was added to the framework.

In Figure 4.1 you will find a diagram describing the changes to the core layout of MARLA that were required to add the DHA module. All components marked with DHA indicate significant changes in order to incorporate Dynamic Heterogeneity Awareness. To this end, the *Initialization* component of the original

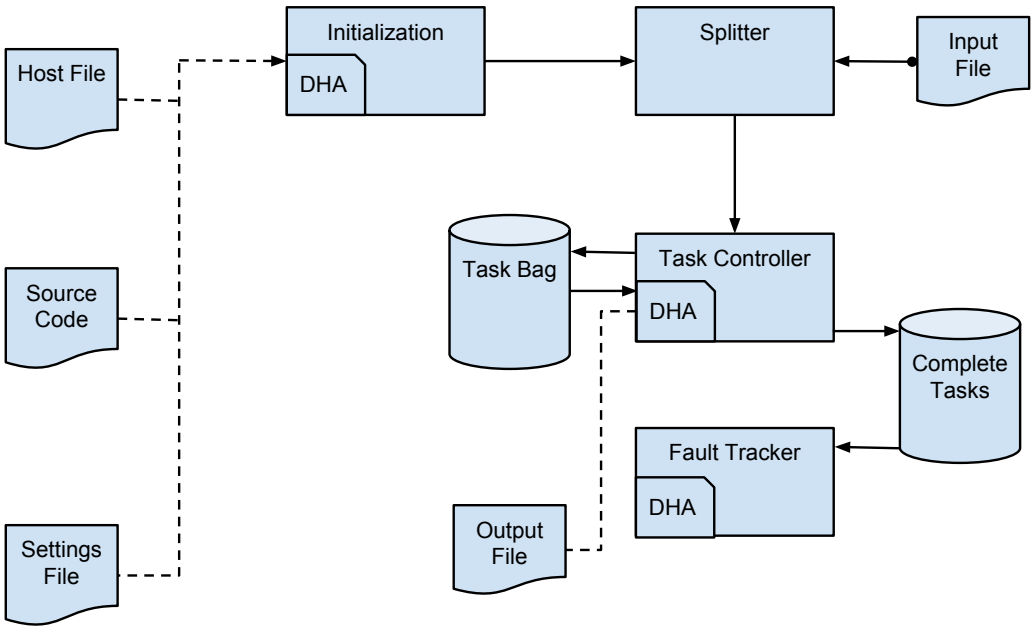

Figure 4.1: Design of MARLA-DHA

MARLA framework is modified to memoize the number of cores available on each host. Additionally, the *Task Controller* and *Fault Tracker* components of the original MARLA have been modified to assign tasks in batches (based on the number of cores available on a worker node, as opposed to a user-defined parameter).

This module operates as follows:

1. When parsing the configuration file for the framework, retrieve from each work in the *HOST* file the number of cores that they have available. This is done by executing the command:

```
cat /proc/cpuinfo | grep processor | wc -l
```

which returns a single integer value, which is then memoized and stored within the framework. Note that a similar technique is used in later works of other top-level Apache Projects for advertising resources in Apache Mesos [35].

2. When processing dividing work amongst the workers, the thread for each host is passed two additional parameters. The first such parameter is the task number at which this worker is to start processing, and the second parameter is the number of tasks follow that number that it should work on. For example, if the first worker to be assigned work were to have four cores, then the first parameter this worker would be passed would be zero (since no other work has been assigned) and the second parameter it would be passed would be four (since it has four cores that can receive work). Similarly, the next host to be assigned work would be passed four as the number task it should start processing and would also be passed the number of cores on that host.

3. Once given an assignment, a worker will spawn as many threads as it has cores (it knows this value from the parameter passed in its assignment) and will require each thread to process a single task (whose task ID is also passed via its assignment). The assignments of task IDs are processed in a linear fashion. All cores have a thread spawned, so it is possible that a task ID does not exist, despite a core being assigned that ID. When this happens, the thread will recognize that the task ID is invalid and will terminate itself.

While this process eliminates the need for a user-defined, easily misconfigured variable to the framework, and allows Dynamic Heterogeneity Awareness via the DHA module, it suffers from several drawbacks.

The first drawback is that work cannot be evenly distributed across the workers when there is a mis-match between the number of cores in the cluster and the number of tasks. For example, if the cluster is made of ten nodes with ten cores each (for a total of 100 cores), and there were 50 tasks assigned for a given workload, the work is distributed only amongst half of the cluster (the first five nodes would each receive ten tasks that they were to process, one for each of their cores). To avoid this problem, one thread would have to be created per task on the master node in order to distribute tasks more evenly. However, this would require the addition of overhead that will be insurmountable at scale. Another option would be to have the workers communicate with one another their task status and trade task assignments in order to keep the work more evenly distributed. This adds new layers of complication that would require the worker nodes to notify the master node that it is defying its assignments, and would add additional network congestion, especially at scale. Instead, we decide to keep the amount of communication between worker nodes at zero and let the assignments of tasks to workers be handled in a simple, linear manner by the master node.

The second drawback of this process is that if a task were to become defunct, that task would have to be re-scheduled in a batch with other tasks that may or may not be defunct. For example, if task 50 were to have an error, it would be registered as a fault in the TaskTracker component of MARLA. When making an attempt to re-assign this task, the DHA Module will assign task 50, and the next

N consecutive tasks to the next available worker (where N is the number of cores on that worker). This may mean that duplicate work is performed in order to recover from this task. However, without this strategy, the worker node will not be fully utilized, and the master node would have a significant amount of additional overhead in managing the specifics of each individual task (in order to schedule one task at a time on each core of each worker). Again, to keep overhead low we decided to let the master assign tasks to workers in a linear manner via the master node.

4.2 Experimental Setup

There are four hosts with the following characteristics (we'll refer to these as Host Type A):

- 24 core - Intel Xeon CPU E5-2620 v3 @ 2.4GHz
- 65.867 GB of Memory

There are two hosts with the following characteristics (we'll refer to these as Host Type B):

- 40 core - Intel Xeon CPU E5-2650 v3 @ 2.3GHz
- 131.92 GB of Memory

There are two hosts with the following characteristics (we'll refer to these as Host Type C):

- 32 core - Intel Xeon CPU E5-2640 v3 @ 2.6GHz

- 65.867 GB of Memory

Additionally in these experiments, the number of tasks was increased from 96 (the smallest number of cores in any configuration) to 720 (3 times the largest number of cores in any configuration).

To compare configurations, we assign a value to the configuration of the cluster as follows:

$$CONFIG = 2.4 * 24 * A + 2.3 * 40 * B + 2.6 * 32 * C$$

In this case, A is the number of machines of Type A, B is the number of machines of Type B, and C is the number of machines of Type C. Note that the calculation can also be expressed as the sum, over each type of machine `m` in the set of machines in the cluster `M`, of the product of `m`'s clock speed, `m`'s number of cores, and the number of machines of type `m` in a given configuration.

$$CONFIG = \sum_{m \in M} ClockSpeed_m \times Cores_m \times Count_m$$

It's worth noting that for the purposes of these experiments, there are not any duplicate values for the configuration settings, though one mechanism for accommodating that in larger clusters would be to also consider measurements like TDP (Thermal Design Power) for each type of chip, or to also consider available memory on the machines. The configuration values for these clusters are detailed in Table 4.1.

4.3 Results

4.3.1 File Size Impacts

Refer to the results of the Figure 4.2. In this set of experiments the file size was held constant at 1000MB and the configuration of the cluster was modified such that all combinations of nodes are run. As we can see, the appearance of a converging wave appears as the number of tasks increases. We can also see that the configuration increases (the cluster becomes more powerful), the wave shrinks in amplitude. This is because as the cluster performance increases, the amount of time spent processing is reduced (as the tasks are able to be processed more quickly). This is in opposition to the findings from MARLA without the DHA Module, which was not as readily able to adapt, especially when the `CORES_PER_WORKER` variable was mis-configured. Figure 4.2 also shows that there is overhead associated with adding too many tasks. This is visualized by the upward sloping trend as the number of tasks increases.

Comparing file size of 1000MB (as seen in Figure 4.2) and 3000MB (as seen in Figure 4.3) with various configurations, we can see that the converging wave pattern still exists for larger file sizes. With three times as much data, the difference in execution time afforded by successive upgrades to the cluster is less prominent, however, the results in Figure 4.3 show that the proper setting of tasks is still important for reducing turnaround time of the application.

In Figure 4.4 we only consider two configurations 324.8 (1 A type node, 2 B type nodes, and 1 C type node) and 350.4 (no A type nodes, 2 B type nodes, and 2 C type nodes), and a fixed file size of 4000MB. This difference between these two

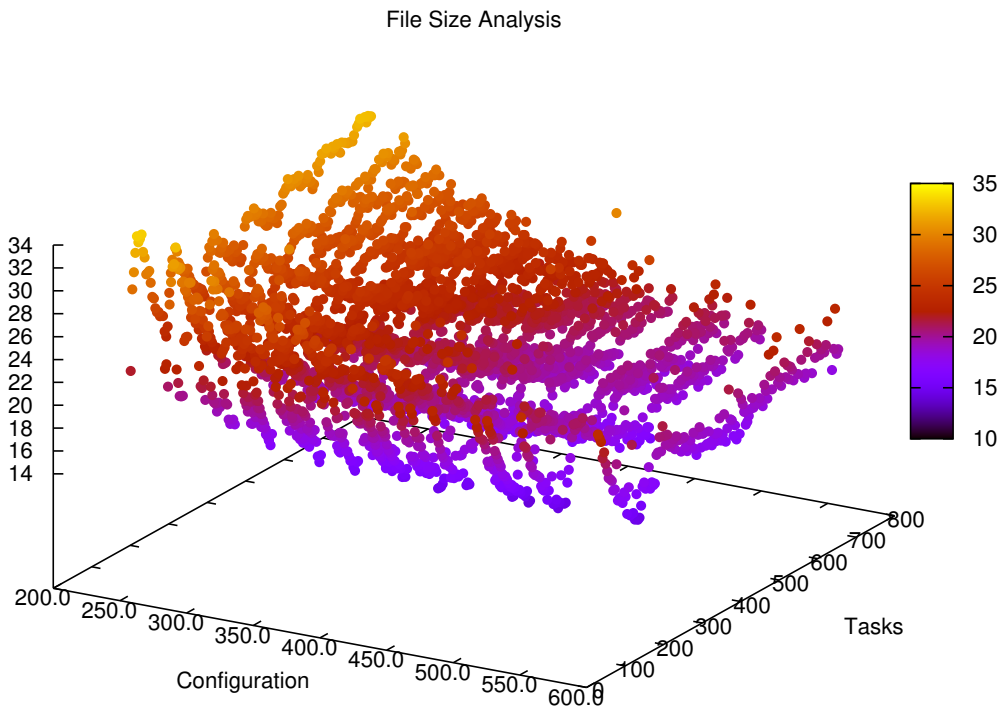

Figure 4.2: Execution time for the task and configuration splits when file size is held constant at 1000MB.

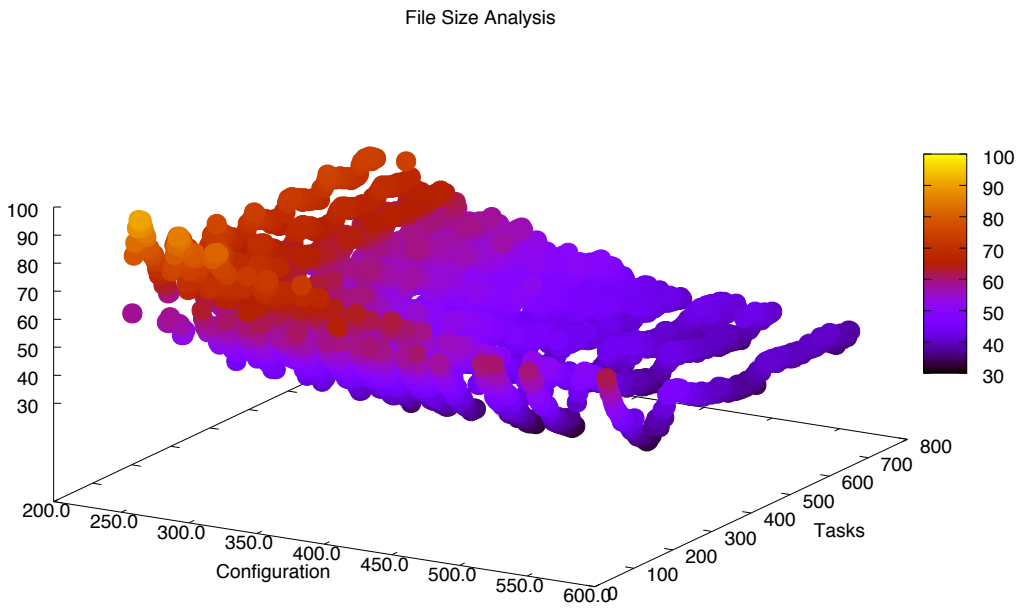

Figure 4.3: Execution time for the task and configuration splits when the file size is held constant at 3000MB.

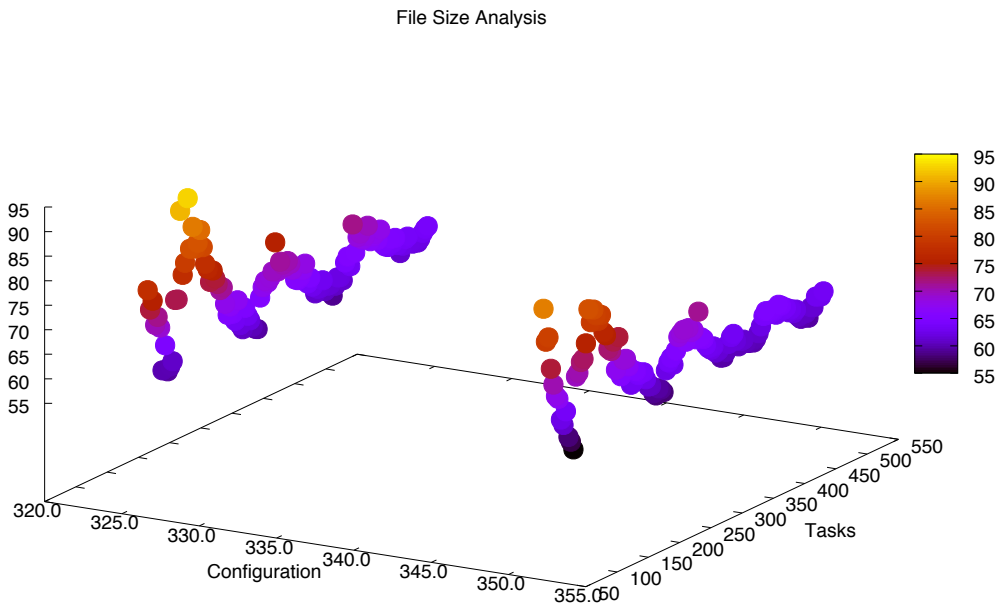

Figure 4.4: Execution time relative to task count with fixed file size of 4000MB, and configurations that represent upgrading a type A node to a type C node.

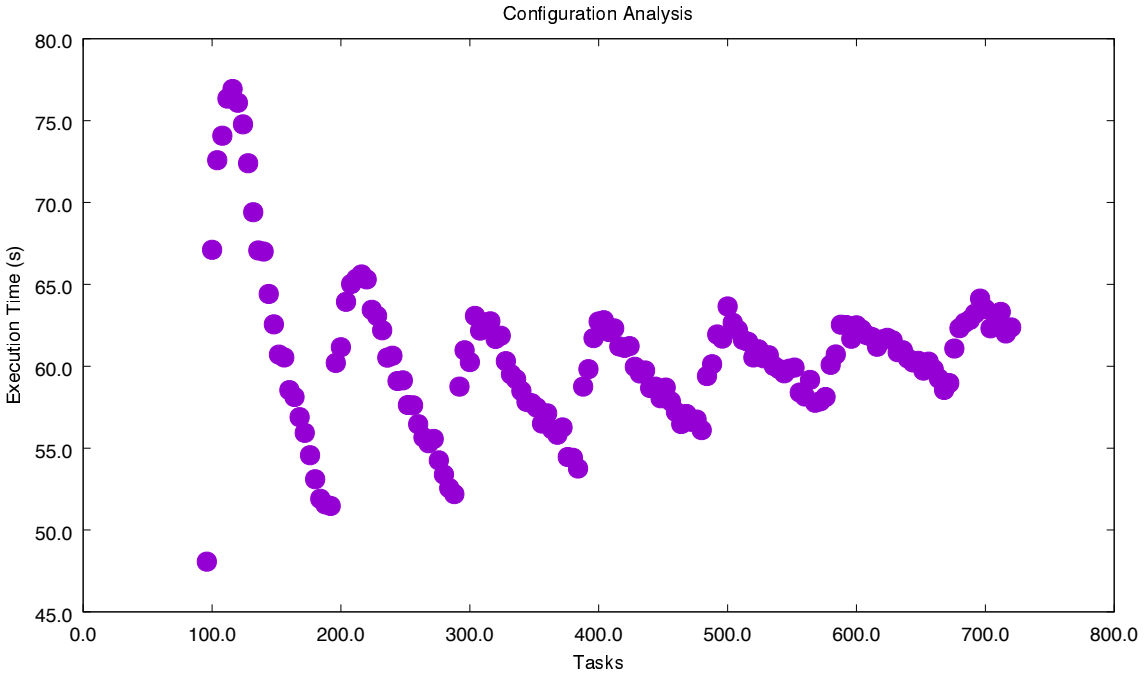

Figure 4.5: Execution time for configuration 230.4 (4 nodes of type A), versus number of tasks for file size 2500MB.

configurations illustrates what happens when an A type node is upgraded to a C type node. Notably, in this graph we can see that the upper bound on execution time decreases, but the lower bound on execution time remains approximately the same. Notably, the upgrade does afford us more task settings that are near-optimal in terms of turn around time. This means that by performing an upgrade, we don't necessarily have to change the number of tasks to be able to reap the benefits of upgrading as few as one node.

Figure 4.5 shows the convergent wave pattern, as seen before. This is true, even as there is limited heterogeneity in the cluster, since the platform and hardware are

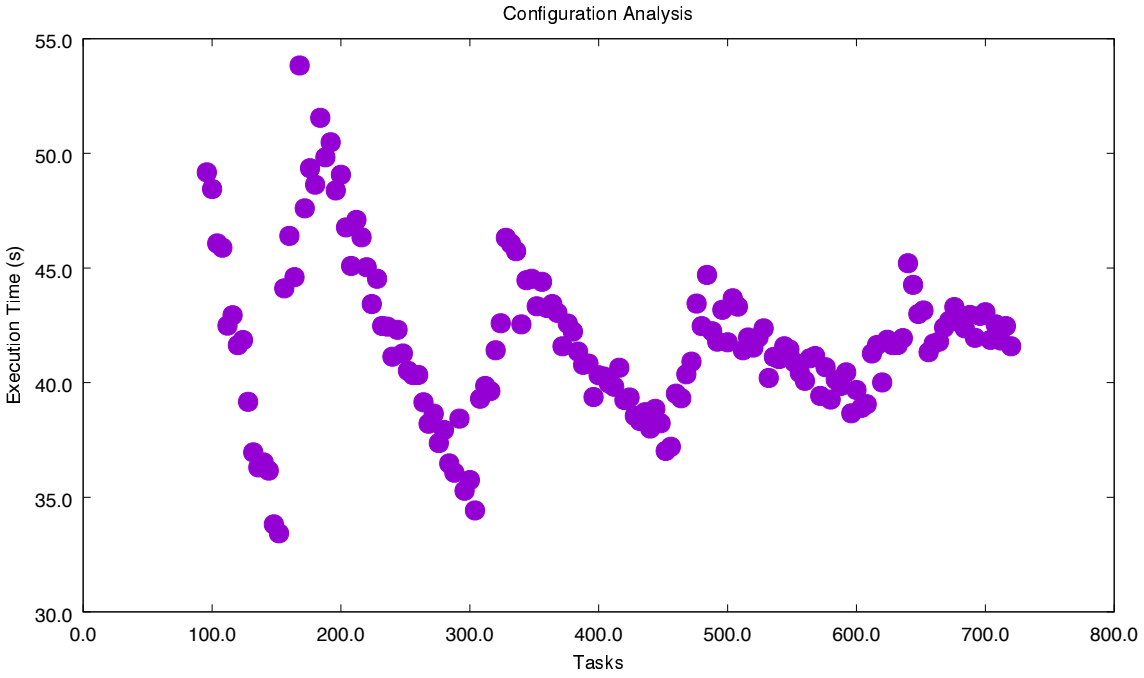

Figure 4.6: Execution time for configuration 356.8 (3 nodes of type A, 2 nodes of type B), versus number of tasks for file size 2500MB.

the same, leaving only dynamic heterogeneity to be mitigated. In an effort to understand more of how upgrading the nodes in the cluster leads to the need for different configurations, we consider alternative configurations as well. In Figure 4.6 we consider configuration 356.8, which consists of 3 nodes of type A and 2 nodes of type B. Notably as the cluster configuration becomes more complex the variability of the performance becomes less predictable, particularly as the number of tasks increases. Similarly to the variance we see between the data points in Figures 4.5 and 4.6, the data continues to have additional variance as the configurations get more complex, as seen in Figure 4.7.

Another class of experiments looks at the configurations of the cluster as we vary the file size and the cluster configuration. This analysis (as seen in Figures 4.8

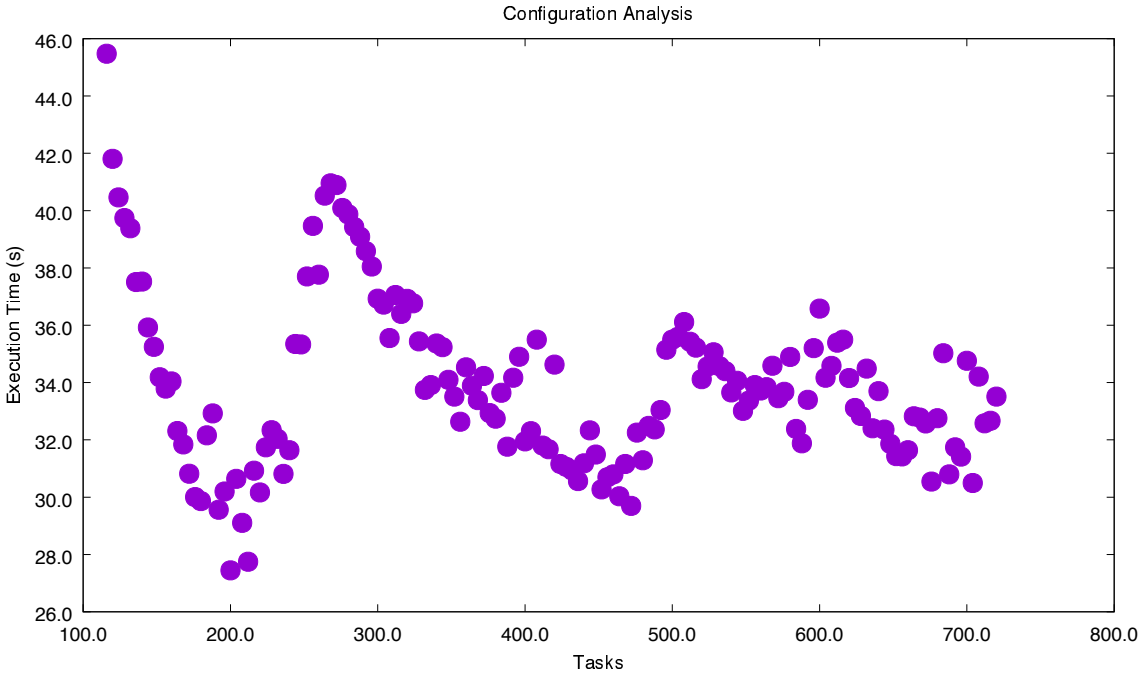

Figure 4.7: Execution time for configuration 580.8 (4 nodes of type A, 2 nodes of type B, and 2 nodes of type C), versus number of tasks for file size 2500MB.

and 4.11 shows that the convergent wave pattern as the number of tasks increases is held more closely (and more predictably) when there is fewer variations among the classes of worker nodes in the cluster. For example, the pattern is displayed clearly in Figure 4.8 but the pattern becomes fuzzy and has more deviation in Figure 4.11. The pattern does indicate that the lowest execution time occurs when there is the least amount of overhead (i.e. where the number of tasks to be processed is equal to the number of cores in the cluster). Similarly to this, we can see from Figure 4.10 that the fuzziness of the wave pattern (and its variances from the expected flow of the wave), begins to get worse as more tasks are generated and spreads to where less tasks are generated. In particular, this means that as long as the expected number of tasks to be assigned to each core is not sufficiently large (3 or more) then the execution time is more predictable and does not depend as heavily on the diversity of the nodes in the system.

Execution time for the tasks as the number of tasks increases. The graph shows that the convergent wave pattern as seen before is also demonstrated in this instance. The cluster configuration represented has a value of 414.4, which represents 4 nodes of type A, 2 nodes of type B, and no nodes of type C, making the difference between this configuration and the one in Figure 4.9 is the addition of a single A type node.

The graph in Figure 4.11 shows the convergent wave pattern as seen before is also demonstrated in this instance. Note that the difference between this figure and Figure 4.8 shows that as the cluster configuration gets more complex (more

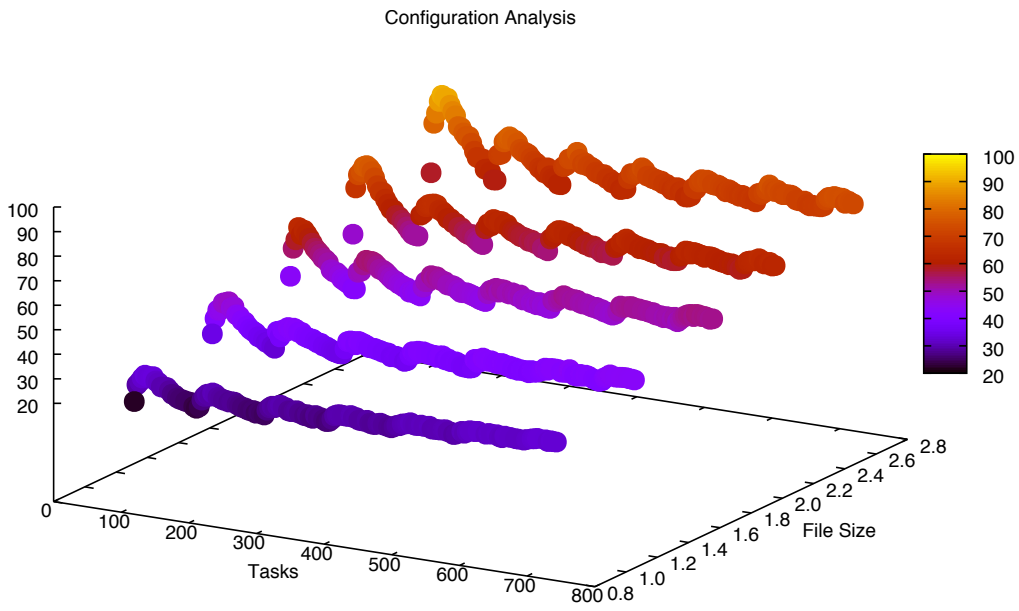

Figure 4.8: Execution time relative to number of tasks for configuration 230.4 (4 nodes of type A) for five different file sizes: 0.9, 1.4, 1.9, 2.3, and 2.8 GB

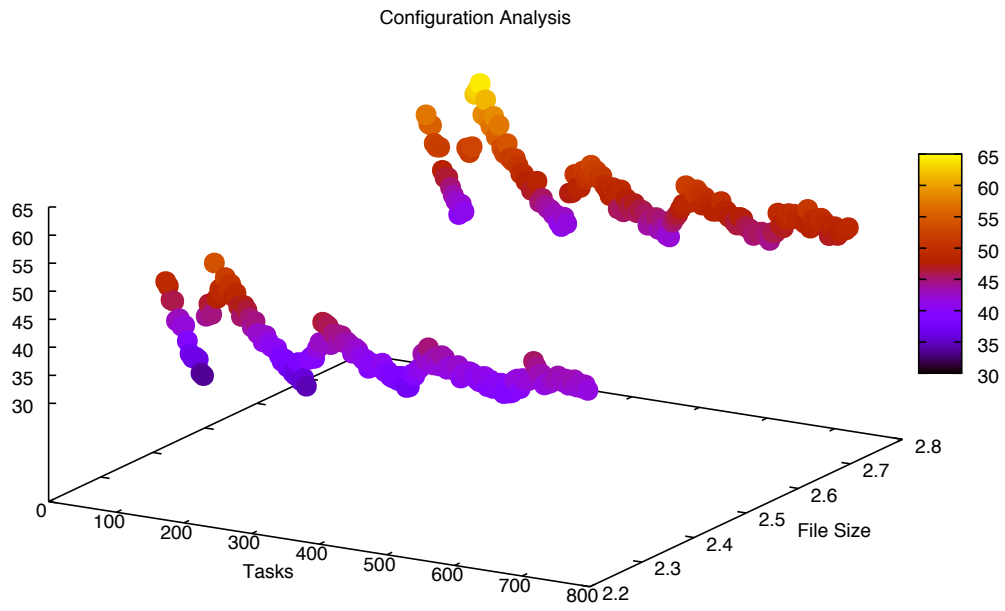

Figure 4.9: Execution time relative to number of tasks for configuration 356.8 (3 nodes of type A, 2 nodes of type B) for two different file sizes: 2.3, and 2.8 GB

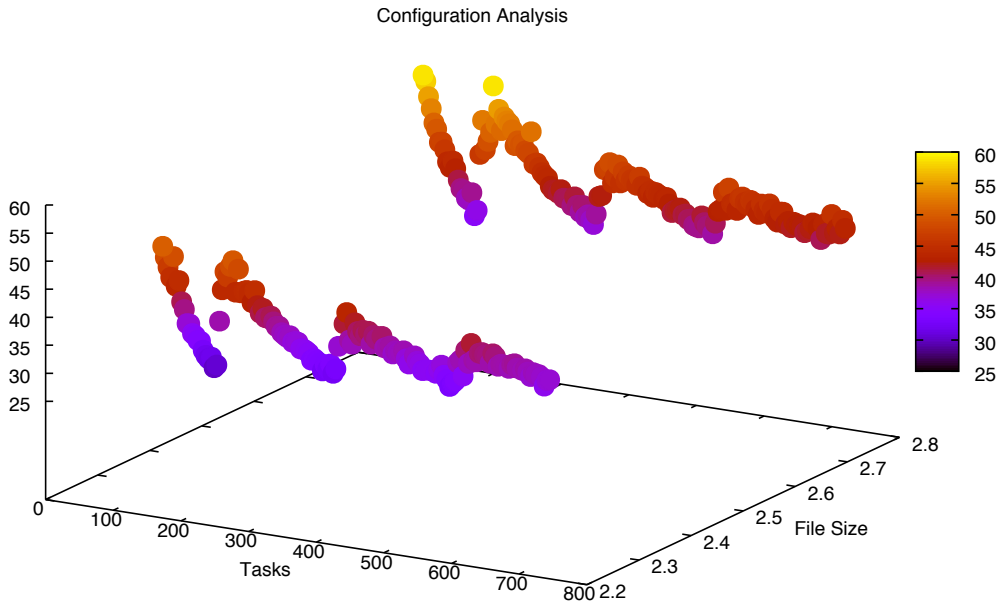

Figure 4.10: Execution time relative to number of tasks for configuration 414.4 (4 nodes of type A, 2 nodes of type B) for two different file sizes: 2.3, and 2.8 GB

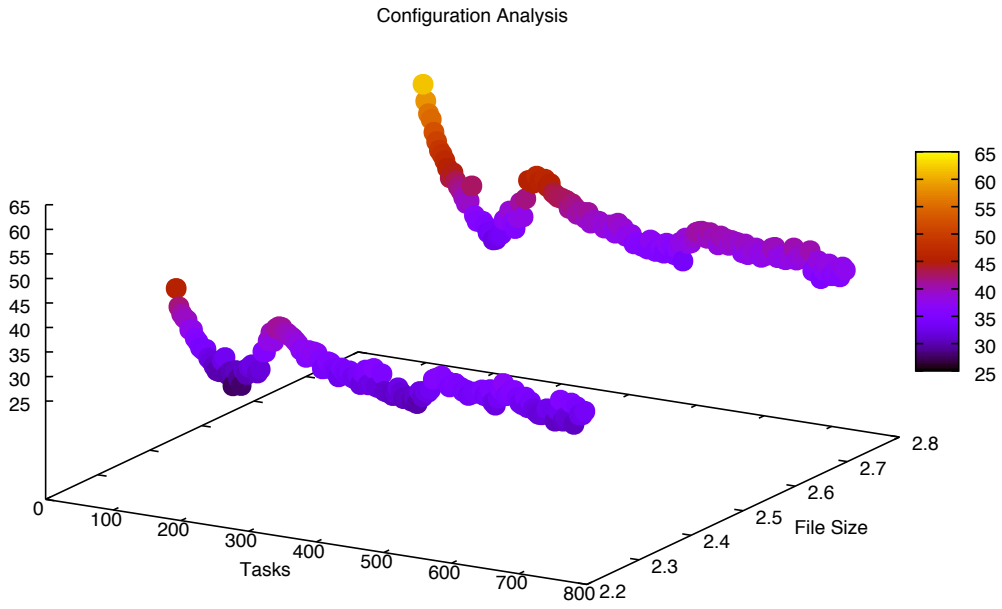

Figure 4.11: Execution time relative to number of tasks for configuration 580.8 (4 nodes of type A, 2 nodes of type B, and 2 nodes of type C) for two different file sizes: 2.3, and 2.8 GB

hosts and more host types) the variability of the performance becomes less predictable. The cluster configuration represented has a value of 580.8, which represents 4 nodes of type A, 2 nodes of type B, and 2 nodes of type C.

All things in this chapter considered, we can tell that as a cluster increases in variance of the type and capabilities of nodes, the following becomes more important:

- File Sizes that are large help to offset the costs of returning back to the master node for more work.
- Number of Tasks that are significantly larger than the number of nodes in the cluster allows for more wiggle-room when it comes to performance despite a sub-optimal configuration.

Notably, as the heterogeneity of the cluster increases, it is ever more important to have a sufficient number of tasks that they may be distributed throughout the cluster really drags down the turnaround time if there is no speculative execution, which is something missing from this framework.

Table 4.1: Configurations of the Cluster

Count(A)	Count(B)	Count(C)	Configuration
0	2	2	350.4
1	1	2	316.0
1	2	1	324.8
2	0	2	281.6
2	1	1	290.4
2	2	0	299.2
3	0	1	256.0
3	1	0	264.8
4	0	0	230.4
1	1	2	408.0
2	1	2	373.6
2	2	1	382.4
3	0	2	339.2
3	1	1	348.0
3	2	0	356.8
4	0	1	313.6
4	1	0	322.4
2	2	2	465.6
3	1	2	431.2
3	2	1	440.0
4	0	2	396.8
4	1	1	405.6
4	2	0	414.4
3	2	2	523.2
4	1	1	488.8
4	2	1	497.6
4	2	2	580.8

Chapter 5

Co-tenancy

Historically there have been number of Big Data platforms available for use on a cluster. In order for companies or academic institutions to leverage each of these Big Data technologies, they would need to statically partition their clusters and set up each partition with one such framework. We shall start by exploring a few of the key frameworks in this space and discuss their evolution.

5.1 Hadoop

Hadoop was the first big player in the Big Data landscape, with the introduction of MapReduce to the academic and general computing landscape [21], Hadoop clusters of all sizes, from tens of nodes to thousands of nodes, began to spring up in data centers and labs around the world [3]. Traditionally Hadoop clusters are just that, clusters used only for Hadoop. The drawback here is that the resources not being used for Hadoop jobs at any given time would not be able to be leveraged by other jobs that need to run on a distributed cluster. Notably, Hadoop was built for

batch processing of large data sets, not for processing of streams or even batch processing of small data sets. In fact there has been a lot of analysis into resolving or trying to provide multiple solutions to the problem of storing and processing small files on Hadoop, but the nature of the Hadoop framework and HDFS in general is such that this is not considered wise to continually write, consume, and leave in tact small files [25, 14, 83]. These issues helped to highlight that as Hadoop clusters would get larger and larger (as required for growing data sets), their compute resources may end up going to waste.

5.2 Spark

Apache Spark is used for processing data quickly in large scale distributed systems by focusing on data science and abstractions [80]. Spark uses in-memory data processing to provide performance that can be 100x faster than Hadoop. Spark was designed to support more classifications of applications on top of HDFS than just MapReduce, and is generally considered to be the next-generation of MapReduce [66]. Spark introduced the abstraction of Resilient Distributed Datasets (RDDs) to support running applications more efficiently. RDDs are stored in memory without replication, RDDs are mutated (via map, join, groupby, etc.) and each such transformation is stored alongside the RDD. The core of Spark is the implementation of these RDDs, where each RDD is a Scala object that is constructed in one of four ways:

- From a file in a shared filesystem
- By dividing a Scala collection into a number of slices to be sent to multiple

nodes

- By transforming an existing RDD (via flatMap, map, and filter)
- By changing modifying state of an existing RDD through actions:
  - The $cache$ action suggests that the dataset should remain in memory as it will be reused
  - The $save$ action evaluates the dataset and writes it back to the distributed filesystem, with the saved version being used in later operations on the RDD

Spark has shown that traditional MapReduce afforded by Hadoop has several drawbacks and is not readily capable of leveraging the Hadoop MapReduce infrastructure. Notably Spark has been expanded with multiple plug-ins that interact with the core API to facilitate interaction with SQL and DataFrames, Streaming applications, Machine Learning libraries, and Graph Computation. Spark itself is able to run standalone or as a framework on other distributed systems.

5.3 MPI

Still in use today, MPI continues to be a platform for processing Big Data [10], with its interface for communication between processes, and ability to be finely tuned, this framework runs more than just legacy Big Data applications.

OpenMPI [79] has three abstraction layers that afford it the flexibility to have different types of applications (following no strict paradigm, as opposed to Hadoop) successfully be programmed using the well-defined APIs. These layers are:

- $OPAL$ - Open, Portable Access Layer - This is the bottom layer of the MPI stack and it focuses on individual processes (as opposed to parallel jobs). This provides core portability between different Operating Systems, as well as the glue code needed for basic functionality (like generic linked lists, string manipulation, etc.)

- $ORTE$ - Open MPI Run-Time Environment - This is the middle layer of the MPI stack and it provides a runtime system to launch, monitor, and kill the parallel jobs running on the MPI framework. Notably a parallel job is comprised of one ore more processes that are bound and act together as a single unit. In more advanced clusters, ORTE provides specialized APIs so that processes can be launched and regulated across multiple servers.

- $OMPI$ - Open MPI - This is the highest layer of abstraction that implements all of the message passing semantics as defined by the MPI standard.

For the sake of speed and efficiency, each layer is able to bypass the layers below it to interact with the OS and/or hardware when needed. MPI allows for heterogeneous compute workloads to be simultaneously scheduled, and in order to do this relies on managed cluster environments like Torque[54] and SLURM[47]. MPI is well known for its ability to process big data, but requires a lot of setup and sometimes intimate details of the underlying architecture, which makes it difficult to schedule different types of workloads on MPI.

5.4 Co-tenancy in the Big Data Landscape

From Hadoop[3], to Spark[80], to MPI[79, 10], given that each of these platforms has different scheduling needs and differing hardware requirements, the traditional wisdom for setting up a Big Data Processing cluster was to partition the compute resources such that each framework can have its own dedicated resources. This encourages predictability, however, this setup is limiting insofar as the number of resources available to each framework is rigid, and the rigidity can be difficult to workaround as processing of big data applications sometimes require the available resources for each framework to be malleable.

For example, an application that was built on Hadoop for MapReduce is batchy by nature, but as the data sets grow, some of these applications can be built to process streaming data as part of Spark instead, which can provide a realtime element to the computations. In this traditional setup, applications that move from Hadoop to Spark will have to remove resources from the Hadoop partition and add them to the Spark partition.

Similarly, as new frameworks are regularly being developed, each time a new one comes onto the scene, resources will have to be taken from the partitioned frameworks in order to create a new partition for the new framework. Another scenario where this is familiar is with regard to major version upgrades for each of the frameworks running atop the cluster. In these scenarios static partitioning of the cluster is very time consuming and can impede the ability of application writers to performing upgrades as their cluster allocations are too rigid.

As a result of these things the need for tenancy of multiple Big Data processing

platforms across all hardware infrastructure in a data center is seemingly unavoidable. Several solutions were proposed to address this need, however the primary solution involves adding additional layers of abstraction to the frameworks running these applications. This allows for other applications to be run on the same hardware throughout the data center without starving any one framework, and also without having to maintain separate clusters for each such framework.

5.4.1 Hadoop V2

As Apache Hadoop progressed it became more and more obvious that the *JobTracker* service, that was built as part of Hadoop in order to allow MapReduce processing, too closely coupled the Hadoop Ecosystem and MapReduce. Because of the structure of Hadoop V1, when a programming model didn't fit cleanly into the MapReduce paradigm, there was a lot of overhead associated with trying to process data in Hadoop. In order to combat this, the community created YARN (Yet Another Resource Negotiator) to manage resources and provide abstractions that allow for more general-purpose distributed computing [71].

In Hadoop V1 the *JobTracker* was responsible for resource management and job scheduling and monitoring. Instead of this model, Hadoop V2 relies on YARN which has the following components:

- **ResourceManager** - The authority that distributes resources among all applications at the cluster level, acting as the cluster scheduler that schedules based on application requirements. The two main components here are:
  - *Scheduler* - The scheduler allocates resources to running applications using the data structures and policies that were already in place in Hadoop

V1 (queues, capacities, etc.). The scheduler does not monitor or track status of various applications, and does not restart failed tasks, it just schedules based on resource requirements.

  - *ApplicationManager* - Accepts job submissions and launches the first *ApplicationMaster* container for the application running on YARN. Also monitors the *ApplicationMaster* container and restarts it if it fails.

- **ApplicationMaster** - The per-application, framework-specific, entity that negotiates and coordinates the resources available with the *Scheduler*, tracks container status, and monitors container progress.

- **NodeManager** - The per-machine process that is responsible for launching containers, monitoring their usage, and reporting it to the ResourceManager.

Notably, the move to HadoopV2 and YARN allows for other applications to be run alongside MapReduce applications in the same Hadoop ecosystem. YARN frameworks operate under a "pull" model with respect to resources, explicitly requesting well-defined resources from the *ResourceManager*, who grants those requests based on cluster status and fair scheduling practices.

YARN has also been the subject of scheduling research that attempts to decrease makespan of jobs, and increase CPU and memory utilization. This was explored by the Performance Fairness Scheduler for YARN written by Wang and Huang [73]. Additionally, these same goals were explored by the HaSTE scheduler developed by Yi et. al [77]. Other work by Liu et. al aims to shift the paradigm of YARN from scheduling only large batch jobs using their Fair Sojourn Protocol in YARN, their aim is to scheduling variable types of jobs, and this results in a more responsive

cluster when the cluster is operating under significant workloads [48].

5.4.2 Borg

Google has described its cluster management system Borg [72], and its kindred scheduler Omega[64]. The role of this application is to manage a multitude of jobs, belonging to a myriad of applications, across a plethora of machines. Users interact with Borg by submitting their jobs (which consist of one or more tasks) that all run the same binary. Each job runs in what's called a *Borg cell*, which is a collection of machines that are treated as a unit. In order to execute the jobs on the cells, Borg makes use of the following components:

- *Borgmaster* - This is a centralized controller that consists of two pieces:
  - *Borgmaster Process* - This process is responsible for handling RPCs from a client which provide accessor/mutators for the Finite State Machine that represents the lifecycle of an application. The Borgmaster Process is also tasked with managing the FSMs of the entire system, communicating with Borglets, and hosting a webUI.
  - *Scheduler* - Scans the pending queue of tasks and assigns tasks to machines if they can fit. The scheduler uses priority to decide which tasks to schedule first, and checks against feasibility measures and scoring to decide where to place the task. The scoring model described in the literature is one that tries to eliminate the amount of resources that cannot be used because another resource on the machine is fully allocated. This is similar to a heuristic-based multi-dimensional bin-packing problem.

- *Borglet* - This is the agent that runs on every machine in the cell and is responsible for starting and stopping tasks, restarting them on failure, handles monitoring of resources, and reports metrics back to the Borgmaster. The Borgmaster polls the Borglets to get the current state of the machine and send it outstanding requests. If the Borgmaster polls a Borglet and it does not respond for a number of consecutive attempts, the tasks that were scheduled on that node are scheduled elsewhere in the cell, as the worker is considered down.

Borg and YARN have similar goals, to allow applications and frameworks to operate within the same large-scale cluster by providing a layer of abstraction and coordination that is asynchronous to the actual task that needs execution.

5.4.3 Mesos

Coming out of the Berkley Amp Lab [35], and now a top-level Apache project, Mesos operates under a similar premise to YARN, by providing a different layer of abstraction for negotiation of resources in a cluster, frameworks can run co-tenant with one another.

The Mesos Architecture consists of a few components:

- *Mesos Master* - Accepts Framework registrations and decides how many resources to offer to each framework depending on its priority and role.

- *Mesos Slave (Agent)* - Running on each worker node, this component is responsible for advertising the node's resources to the *Mesos Master* so that they can be made into Offers and available for the *Frameworks*.

- *ZooKeeper Cluster* - Used for storing state needed by the *Mesos Master* in order to achieve High Availability.
- *Framework* - Each Mesos Framework consists of two components:
  - *Scheduler* - Registers with the *Mesos Master* in order to be offered resources, and determines which of the Mesos Offers to use for given tasks.
  - *Executors* - Launched on a *Mesos Slave (Agent)* to run the tasks that the framework needs to run.

Mesos has fast become the industry standard for co-tenant applications, and has support for many frameworks including Hadoop V2 via Myriad [9], Apache Storm [37], Apache Spark [80], Apache Kafka [65], Cassandra [13], and other frameworks. Because of its powerful offer abstraction layer, Mesos can even be used to bridge physical servers, virtual servers, and private and public clouds to make a super-cluster. Mesos is widely accepted in the industry across companies like Apple, Netflix, and more [62].

Marathon

Mesos by itself is insufficient in large clusters as there are many types of applications that may need to be run on top of the Mesos Framework, many of which are incompatible with Mesos APIs by themselves. In order to combat this, a general purpose framework named Marathon was developed[51]. Marathon is able to orchestrate both applications and frameworks that run on Mesos. Marathon intercepts all resource offers from the Mesos master and itself acts as a distributor of

offers to these other apps/frameworks (thus acting as a meta framework). In this way, Marathon can act as a container orchestration platform and provide scaling and self-healing.

Marathon, running on top of Mesos, has all of the Mesos core components and they run in the exact same way as they do sans Marathon, however, Marathon itself has some additional components:

- *Marathon Scheduler* - Receives offers from the Mesos Master, and offers them to the applications and frameworks running under Marathon's purview.
- *Docker Executor* - Receives tasks from the Marathon scheduler (either a standalone task or a framework task) and executes them inside of a Docker container on the Mesos Slave/Agent nodes.

Marathon is designed for keeping apps and frameworks running indefinitely, allowing them to be launched on either a static or dynamic set of machines within a Mesos cluster. Since Marathon has constraint management mechanisms, it is able to run frameworks that previously needed dedicated hardware (i.e., Kafka), and can leverage spare resources on those clusters for additional tasks.

Because Marathon only keeps tasks running in a Mesos cluster, submitting a task for Marathon to manage is simple and requires few specifications.

Aurora

Marathon is designed to allow frameworks and applications to be run indefinitely on a Mesos cluster. However, this is not always the desired behavior as sometimes tasks that need to be run need only be run once, or periodically.

Aurora leverages a similar architecture to Marathon and improves on it by enabling a cron-like scheduling capability, allowing tasks to be scheduled periodically. Similarly to Marathon, Aurora is a meta-framework and intercepts all Mesos Offers, distributing them to frameworks and applications based on its own scheduler to the frameworks and applications running under its purview[8].

Aurora has its own Domain Specific Language for defining how and when to schedule tasks. This allows for additional capabilities beyond what Marathon affords, as the expense of increased complexity in defining jobs.

Kubernetes

Kubernetes[40] is designed as a system that automates deployment, scaling, and management of applications that have been containerized. Kubernetes, coming out of Google, has a similar architecture to Borg (discussed in Chapter 5.4.2), and has the following key components:

- *Master* - Places container workloads into pods on the nodes in the cluster. The Kubernetes master is made up of several other components:
  - *API Server* - This is how the Kubernetes master node communicates with the various components and where cluster health information is maintained. Namely, this component relies on `etcd` to store configuration data which can be accessed by this server via HTTP or JSON APIs.
  - *Controller Manager* - This component is responsible for scaling the workloads up and down by comparing the clusters current state and the desired state.

  - *Scheduler* - This component is responsible for making sure that a workload is placed on an appropriate node.

- *Kubelet* - This receives pod specifications by interacting with the *API Server* and manages pods that are running on the same host. This element runs on the master node and on each worker node. For reference, a pod is a group of containers that run on the same node and share resources.

Notably Kubernetes can run atop Mesos[41], and has similar goals to Marathon and Mesos when run together, which comprise DC/OS [74].

5.5 Mitigating Heterogeneity with Co-Tenancy

As referenced in Sections 5.4.1, 5.4.2, and 5.4.3, powerful abstraction layers allow for bridging different kinds of distributed systems, which can also create more heterogeneous clusters than had ever been possible before. As a result, the lessons learned from previous chapters are applicable here, namely that efficient scheduling of tasks is difficult in heterogeneous clusters, and that it pays to know a bit about the workloads that are running before trying to schedule them, as the ideal configurations are not always known. Also worth noting is that the benefits of Kubernetes, Marathon, and the like make it such that there are now a plethora of smaller tasks that can be leveraged for efficiently bin-packing machines with useful work, increasing cluster utilization.

5.6 Co-tenancy Moving Forward

Since co-tenancy is the future of distributed computing and has produced promising results for increasing cluster utilization, we will spend the remainder of this dissertation discussing how the work discussed in previous chapters has already contributed to the distributed computing landscape, even outside of MapReduce. We also discuss ways in which this work can be extended further still.

Intel has included Running Average Power Limit (RAPL)[60] in many of its new chips. RAPL provides a set of hardware counters that provide energy and power consumption information. Utilizing these counters, we will explore the viability of scheduling workloads with this information. In particular, we will set a power budget amongst all worker nodes and use the RAPL information to reduce the impact of worker nodes when the cluster exceeds a user-defined power cap. As hardware in data centers has evolved, RAPL technology provides the ability for users to monitor energy and power consumption on-chip. This is similar the need for temperature-based measurements used in Chapter 2 in order to estimate power usage. Leveraging RAPL for estimating power usage of the whole system allows the integration of power usage into meta-schedulers described in Chapter 5.4.3 that are usable within the Mesos Ecosystem.

In extension work that relies on ideas explained in Chapter 2, a meta-scheduler similar to Aurora's was created to consider adding and removing nodes from a Mesos cluster based on their power consumption using RAPL measurements. This work is described in the paper titled, "Electron: Towards Efficient Resource Management on Heterogeneous Clusters with Apache Mesos" [22]. In this paper, Electron is described as a meta framework, similar to the role that Aurora [8] takes in a

Mesos cluster, this means that Electron acts as a unified scheduling layer through which all Mesos offers are passed. This work considers two main efforts to schedule tasks: first fit and bin packing. The first fit algorithm iterates over the queue of tasks, scheduling them on a first come first served basis during each round of offers that come through Mesos. As a result, when offers are made to the Electron framework, an offer is consumed as soon as it is able to fit a task in the queue. The alternative scheduling model is to use bin packing. The bin packing algorithm iterates over the queue of offers, scheduling as many tasks as will fit in the offer before giving the remnants of that offer back to Mesos. The Electron model tries to limit the amount of power used by the cluster by capping the power consumption of the nodes in the cluster using features enabled by RAPL[60]. The framework uses throttling of power consumption on a per-node basis. This is akin to the methodology used in Chapter 2 where nodes that are presumed to be using too much power will have their contribution of power limited. However, the flexibility of RAPL allows that instead of nodes being removed entirely from the set of nodes that are able to process work, these nodes are instead throttled to consume less power.

In the Electron framework, there are two power capping strategies analyzed. The first strategy is Static Power Capping that sets the loose upper bound of the worker node's power consumption to half of the Thermal Design Power (effectively limiting the node to half of its maximum throughput). The second power capping strategy is Dynamic Power Capping which relies on historical power usage. The methodology applied here is that an Average Historical Power usage of the cluster is maintained, and when the threshold exceeds a high threshold then the node that is consuming the highest amount of power is capped to half of its

Thermal Design Power (again, limiting the node to half of its maximum throughput). When the cluster has recovered, and the Average Historical Power usage falls below a low threshold, the node that was last capped is uncapped, and work proceeds. In the paper, the Electron framework is analyzed by using a series of Docker-ized [50] benchmarks from DaCapo [12], Phoronix [59], Mantevo's MiniFE [52], and NERSC's `Stream` and `Dgemm` benchmarks [56].

The framework shows that using power consumption of a cluster can be kept within a desired range by leveraging newer tools such as RAPL to make scheduling decisions. Similarly this work relies on deferred binding of tasks in order to keep scheduling options open, and in fact, with the power capping strategies relies on dynamically responding to the state of a single node in the cluster to make scheduling decisions. As an extension to this work, another iteration of Electron was discussed in, "Exploiting Efficiency Opportunities Based on Workloads with Electron on Heterogeneous Clusters" [23], with particular focus on heterogeneous clusters. In this work, as with the former work [22], the method of trying to reduce coincident peak power by power capping and therefore adjusting the selection of resources on which tasks can be scheduled, is again relying on deferred binding of tasks and the adaptations necessary to keep the cluster within a fixed envelope of power consumption.

In the extension work, the next iteration of Electron continues to consume offers in a First Fit and Bin Packing strategy, it is expanded to also consider two new strategies: Max Mins and Max Greedy Mins. Max Mins relies on sorting tasks into a deque (double-ended queue) by their Median of Medians Max Power Usage rating in order to define their expected power consumption. In this algorithm, when

deciding which tasks to assign to an offer, the tasks are taken first from the front of the deque, and then from the back of the deque, and this process is repeated until an offer is consumed. In this way, the expected impact of the tasks is mitigated and multiple intensive tasks are less likely to be co-located. Max Greedy Mins on the other hand, while still scheduling tasks in a deque sorted by the Median of Medians Max Power Usage rating, will select one expected high power consumption task from one end of the deque, and then will complete consumption of the offer by taking as many expected low power tasks from the other end of the deque as possible.

Additionally, the extension work forgoes static power capping in favor of progressive extrema power capping. This new power capping model is triggered when the cluster exceeds the power threshold set by the power envelope. Once the algorithm has been triggered, it starts by picking a victim node, selecting the node with the highest average power consumption. To cap, in each iteration, a node is power capped by reducing its current cap by half. In this way, nodes that are repeat offenders are able to quickly have their power consumption adjusted. Additionally, once the cluster falls below the low power threshold, then the node with the highest cap can be uncapped, again by a factor of two, and so nodes that are more aggressively capped are able to be more aggressively uncapped.

From Electron and its successor, it is evident that the work in this thesis has informed the path for an analysis on the adaptability of a bag-of-tasks mechanism for use in shared and heterogeneous systems (i.e., clusters that use Mesos to run multiple frameworks simultaneously). Notably, in each of these frameworks the bag-of-tasks is ordered in an effort to establish precedence related to scheduling,

and further work is done to explore the viability of responding to this precedence, whereas the work contributed by this thesis assumed that all tasks had equal precedence.

5.6.1 Ever Shifting Landscape

As growth in the big data landscape is changing, and as data sets continue to grow, we continually find ourselves on the edge of the next great thing, where there are more areas for scheduling and the compute infrastructure is always evolving. Each of these areas is an opportunity for growth and new challenges, where we can take some of the lessons learned in the past and apply them to new environments, and some situations in which we have to adopt entirely new techniques for scheduling. Below are some of those new environments and the challenges they pose:

- Hybrid Cloud - Many companies and research institutions support on premises data storage and processing. In the same vain, many companies and research institutions also support cloud-based storage and processing. As these entities shift back and forth between these two extremes, they exist on a spectrum of Hybrid Clouds: a combination of public and private clouds[46]. Hybrid clouds also provide a specialized opportunity to schedule tasks in one of two locations, each of which has an associated cost. As applications and services are moved from one cloud type to another, the types of scheduling algorithms that are needed in order to make strides toward shifting one way or the other change. While minimizing cost when considering hybrid cloud scheduling has been addressed to some extent by Shifrin et al.[67], there is another interesting, not oft considered mechanism that should be considered

in hybrid cloud scheduling algorithms, like how to reconcile cost and longer-term scheduling goals, like moving some percentage of an entity's compute to public cloud.

- Fog Computing - As the Internet of Things (IoT) connects many devices, there is a need to move compute closer to the edge of where these devices produce data. This area, that connects the cloud and the devices that generate much of the data stored in the cloud is known as the fog. Fog computing has become an area of increasing interest[78]. A natural consequence of the existence of the fog, is applying the MapReduce paradigm of bringing the compute to the data[20]. In an attempt to do this, it makes sense to leverage container orchestration techniques that are actively being developed in order to bring together the scheduling of data processing, the IoT devices that produce the fog, and the containers that allow us to portably of environments in which to process data.

- Big Data Privacy - Another interesting challenge in the landscape of big data applications and scheduling is that new focus has been placed on data anonymization, and being able to disconnect the source of the data (i.e. users) and the place in which the data eventually lands (i.e. data warehouse, Hadoop, etc.). However, anonymization has impacts on the three features of big data: volume, variety, and velocity[68]. And while there are strategies for making data as anonymous as possible, employing these strategies often mean that algorithms that were previously used in a straightforward manner are less able to be used in the same ways, as the structure of data has most likely

had some transformations applied to it to remove the unique set of identifiers such that the data is sufficiently anonymized[24]. This makes room for additional challenges, as the structure and scheduling of previous workflows may need to be adjusted accordingly. Such data challenges may not always be able to be done in an application developer-transparent way, especially if older data stores need to be adjusted or transformed in order to support privacy, as described by Chen in Cheetah[15].

- Big Data Architecture - As the need for big data processing grows, the types of platforms required to process this data become increasingly complex, as noted by Noh and Lee[57]. Big data platforms require that a cluster can collect, store, process, search, and analyze large data sets, among other responsibilities. Each of these things needs has a different footprint, and a different set of constraints when it comes to scheduling. As a result, the needs of big data architectures have extended far beyond the simple MapReduce model that Hadoop used to implement, and in order to have a cohesive platform it is essential that technologies like YARN continue to exist and be leveraged.

5.6.2 Moving Forward

Work continues to be done in the domain of scheduling distributed systems. As time passes, various open source projects have bridged a divide so that scheduling can be done across increasingly heterogeneous clusters, and even more work is being done leveraging some of the concepts described in this dissertation to more effectively schedule work in a cluster. By making dynamic scheduling decisions

to reduce the work performed by slower and/or power inefficient nodes, this dissertation serves as a compendium of the contributions I've made to the field of scheduling in distributed systems, and how it has been used to derive additional solutions to the problem of heterogeneity.

Further ways in which the work in this dissertation can be expanded are as follows:

- Deferred binding of tasks relies on idle resources being available in a cluster such that work can be scheduled onto any available node at any time, as long as the node is not over-burdened. When this methodology is applied also to Mesos there remain open questions about whether or not the deferred binding of tasks is no longer able to be leveraged when there exists more than one meta-framework. To extend this work, a metric to quantify how many tasks are leveraging late binding of tasks can be developed. Once this is developed, work can be done to analyze the impact of multiple meta-frameworks (i.e., Aurora and Mesos) to quantify the impact these frameworks have on the utilization of late binding, and therefore the performance of the cluster.

- One limitation of the RAPL counters[60] is that they are not able to provide per-core power consumption models. As described in Chapter 4 the mapping of tasks to the appropriate number of cores has a dramatic impact on the turnaround time of applications. This work can be extended to see, specifically, how the spreading of work across the cores of a CPU impacts its overall power consumption, and can leverage the advertised resources in a Mesos cluster accordingly.

- Similarly to the last expansion idea, if the per-core utilization is not able to be

considered because the counters are not available, this work can be extended to identify when a node, instead of being capped to reduce power consumption, should have more work scheduled on it in an effort to improve the density of assigned tasks such that nodes that are not densely packed with tasks may be very aggressively capped.

- While this work discusses switching from many tasks that are sub-divided into additional tasks, and circumventing that in order to more effectively manage heterogeneity, an expansion of this would include leveraging the same behavior in Mesos-managed clusters. Namely, if the same set of tasks were to be repeatedly resized with respect to their Mesos ask, power consumption and turnaround time could benefit, and additional analysis can be performed to identify the ideal amount of overhead that should be included in the Mesos ask.

- Mesos relies on Dominant Resource Fairness scheduling algorithms to disseminate offers to its registered frameworks. This is similar to the way that MapReduce frameworks like Hadoop co-schedule MapReduce tasks. Additional offer distribution policies can be added into the Mesos framework, and their impacts analyzed. As an expansion of ideas presented in this work, one such mechanism for distributing the offers would be to give preferential treatment of offers to frameworks with higher cardinality of outstanding tasks. In this way, frameworks would be encouraged to build out more easily adaptable bag-of-tasks approaches to scheduling.

- Another area of exploration that can build off this work is determining the

effects of hyper-threading as a resource and its impacts on scheduling different types of applications in different compute environments, as well as how the features of applications that Saini first identified in "The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications" interact with one another to make Hyperthreading less optimal, and how to decide heuristically when such applications should not be assigned to the same resources[61].

Bibliography

[1] The FutureGrid Resource Project: An XSEDE Resource Provider. URL: https://portal.futuregrid.org/about.

[2] XSEDE. URL: https://www.xsede.org/.

[3] Apache Hadoop. URL: http://hadoop.apache.org.

[4] HDFS. URL: http://hadoop.apache.org/docs/hdfs/r0.22.0/hdfs_design.html.

[5] National Energy Research Scientific Computing Center. URL: http://nersc.gov.

[6] General Parallel File System. URL: http://www-03.ibm.com/systems/software/gpfs.

[7] Faraz Ahmad et al. "Tarazu: optimizing mapreduce on heterogeneous clusters". In: *ACM SIGARCH Computer Architecture News*. Vol. 40. ACM. 2012, pp. 61–74.

[8] *Apache Aurora*. URL: http://aurora.apache.org/.

[9] *Apache Myriad (incubating)*. URL: http://myriad.apache.org/.

[10] Brandon Barker. “Message passing interface (mpi)”. In: *Workshop: High Performance Computing on Stampede*. Vol. 262. 2015.

[11] Luiz André Barroso and Urs Hölzle. “The Case for Energy-Proportional Computing”. In: *Computer* 40.12 (2007), pp. 33–37.

[12] Stephen M. Blackburn et al. “The DaCapo benchmarks”. In: *ACM SIGPLAN Notices* 41.10 (Oct. 2006), p. 169.

[13] Jeff Carpenter and Eben Hewitt. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. " O’Reilly Media, Inc.", 2016.

[14] S Chandrasekar et al. “A novel indexing scheme for efficient handling of small files in hadoop distributed file system”. In: *Computer Communication and Informatics (ICCCI), 2013 International Conference on*. IEEE. 2013, pp. 1–8.

[15] Songting Chen. “Cheetah: a high performance, custom data warehouse on top of MapReduce”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1459–1468.

[16] Yanpei Chen, Laura Keys, and Randy H. Katz. *Towards Energy Efficient MapReduce*. Tech. rep. UCB/EECS-2009-109. Electrical Engineering and Computer Science Department, University of California at Berkeley, 2009.

[17] Yanpei Chen et al. *Statistical Workloads for Energy Efficient MapReduce*. Tech. rep. UCB/EECS-2010-6. Electrical Engineering and Computer Science Department, University of California at Berkeley, 2010.

[18] Condor High Throughput Computing. *The Condor Project Homepage*. 2007.

[19] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[20] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782.

[21] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[22] Renan DelValle et al. “Electron: Towards Efficient Resource Management on Heterogeneous Clusters with Apache Mesos”. In: *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE. 2017, pp. 262–269.

[23] Renan DelValle et al. “Exploiting Efficiency Opportunities Based on Workloads with Electron on Heterogeneous Clusters”. In: *Proceedings of the10th International Conference on Utility and Cloud Computing*. UCC ’17. ACM, 2017, pp. 67–77. ISBN: 978-1-4503-5149-2. DOI: 10.1145/3147213.3147226. URL: http://doi.acm.org/10.1145/3147213.3147226.

[24] Josep Domingo-Ferrer and Jordi Soria-Comas. “Anonymization in the time of big data”. In: *International Conference on Privacy in Statistical Databases*. Springer. 2016, pp. 57–68.

[25] Bo Dong et al. “A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files”. In:

Services Computing (SCC), 2010 IEEE International Conference on. IEEE. 2010, pp. 65–72.

[26] Zacharia Fadika et al. "Benchmarking MapReduce Implementations for Application Usage Scenarios". In: *Grid Computing, IEEE/ACM International Workshop on* (2011), pp. 90–97. ISSN: 1550-5510.

[27] Zacharia Fadika et al. "MARIANE: MApReduce Implementation Adapted for HPC Environments". In: *Grid Computing, IEEE/ACM International Workshop on* (2011), pp. 82–89. ISSN: 1550-5510.

[28] Zakaria Fadika et al. "MARLA: MApReduce for heterogeneous and Load imbAlanced clusters". In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. 2012, pp. 49–56.

[29] PrimeNet Benchmarks (GIMPS). URL: `http://www.mersenne.org/`.

[30] Jessica Hartog, Elif Dede, and Madhusudhan Govindaraju. "MapReduce Framework Energy Adaptation via Temperature Awareness". In: *Cluster Computing* 17.1 (Mar. 2014), pp. 111–127. ISSN: 1386-7857. DOI: `10.1007/s10586-013-0270-y`. URL: `http://dx.doi.org/10.1007/s10586-013-0270-y`.

[31] Jessica Hartog et al. "Configuring a MapReduce Framework for Dynamic and Efficient Energy Adaptation". In: *Cloud Computing, Industry Track 2012 5th International Conference on*. Honolulu, HI, USA, 2012.

[32] Jessica Hartog et al. “Configuring A MapReduce Framework for Performance-Heterogeneous Clusters/”. In: *Proceedings of the 2013 IEEE Big Data 2014 Conference, Research Track*. BigData ’14. Anchorage, AL, USA, 2014.

[33] Jessica Hartog et al. “Performance Analysis of Adapting a MapReduce Framework to Dynamically Accommodate Heterogeneity”. In: *Transactions on Large-Scale Data and Knowledge-Centered Systems* 9070 (Advanced Techniques for Big Data Management Mar. 2015), pp. 108–130.

[34] Bingsheng He et al. “Mars: a MapReduce framework on graphics processors”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 260–269.

[35] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[36] 1000 Genomes: A Deep Catalog of Human Genetic Variation. URL: `http://www.1000genomes.org`.

[37] Muhammad Hussain Iqbal and Tariq Rahim Soomro. “Big data analysis: Apache storm perspective”. In: *International journal of computer trends and technology* 19.1 (2015), pp. 9–14.

[38] Rini T. Kaushik and Milind Bhandarkar. “GreenHDFS: Towards An Energy-Conserving, Storage-Efficient, Hybrid Hadoop Compute Cluster”. In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. Vancouver, BC, Canada, 2010, pp. 1–9.

[39] Tevfik Kosar and Miron Livny. "A Framework for Reliable and Efficient Data Placement in Distributed Computing Systems". In: *Journal of Parallel and Distributed Computing* 65.10 (2005), pp. 1146–1157.

[40] *Kubernetes*. URL: https://kubernetes.io/.

[41] *Kubernetes-as-a-Service Now Available in DC/OS 1.11*. URL: https://mesosphere.com/blog/dcos-1_11-kubernetes/.

[42] Willis Lang and Jignesh M. Patel. "Energy Management for MapReduce Clusters". In: *Proceedings of the VLDB Endowment*. Vol. 3. 2010, pp. 129–139.

[43] Jacob Leverich and Christos Kozyrakis. "On the Energy (In)efficiency of Hadoop Clusters". In: *ACM SIGOPS Operating Systems Review*. Vol. 44. 1. 2010, pp. 61–65.

[44] Shen Li, Tarek Abdelzaher, and Mindi Yuan. "TAPA: Temperature aware power allocation in data center with Map-Reduce". In: *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE. 2011, pp. 1–8.

[45] Heshan Lin et al. "Moon: Mapreduce on opportunistic environments". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010, pp. 95–106.

[46] David S Linthicum. "Emerging hybrid cloud patterns". In: *IEEE Cloud Computing* 3.1 (2016), pp. 88–91.

[47] Don Lipari. *The SLURM Scheduler Design*. 2012.

[48] Yang Liu, Yukun Zeng, and Xuefeng Piao. "High-Responsive Scheduling with MapReduce Performance Prediction on Hadoop YARN". In: *Embedded*

and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on. IEEE. 2016, pp. 238–247.

[49] Aaron McKenna et al. "The Genome Analysis Toolkit: a MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data". In: *Genome Res.* July 2010.

[50] Merkel and Dirk. "Docker: lightweight Linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2. ISSN: 1075-3583.

[51] *Mesos Marathon*. URL: `https://mesosphere.github.io/marathon/`.

[52] Michael A Heroux et al. *Improving performance via mini-applications*. Tech. rep. Sandia National Laboratories, 2009.

[53] Lm-sensors- Linux Hardware Monitoring. URL: `http://lm-sensors.org/`.

[54] Hidemoto Nakada et al. "Design and implementation of a local scheduling system with advance reservation for co-allocation on the grid". In: *Computer and Information Technology, 2006. CIT'06. The Sixth IEEE International Conference on*. IEEE. 2006, pp. 65–65.

[55] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. "Exploiting platform heterogeneity for power efficient data centers". In: *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*. IEEE. 2007, pp. 5–5.

[56] *NeRSC Benchmark Distribution and Run Rules*. URL: `http://www.nersc.gov/research-and-development/apex/apex-benchmarks/`.

[57] Kyoo-sung Noh and Doo-sik Lee. "Bigdata platform design and implementation model". In: *Indian Journal of science and technology* 8.18 (2015), p. 1.

[58] Lesley Northam and Rob Smits. "Hort: Hadoop online ray tracing with mapreduce". In: *ACM SIGGRAPH 2011 Posters*. ACM. 2011, p. 22.

[59] *Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking*. URL: http://www.phoronix-test-suite.com/.

[60] *Running Average Power Limit – RAPL*. URL: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl.

[61] Subhash Saini et al. "The impact of hyper-threading on processor resource utilization in production applications". In: *High Performance Computing (HiPC), 2011 18th International Conference on*. IEEE. 2011, pp. 1–10.

[62] *Scaling Mesos at Apple, Bloomberg, Netflix and more - Mesosphere*. URL: https://www.linkedin.com/pulse/scaling-mesos-apple-bloomberg-netflix-chuck-taylor.

[63] Michael C. Schatz. "CloudBurst: Highly Sensitive Read Mapping with MapReduce". In: *Bioinformatics*. Vol. 25. 2009, pp. 1363–1369.

[64] Malte Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[65] Abhishek Sharma. *Apache kafka: Next generation distributed messaging system*. 2015.

[66] Juwei Shi et al. “Clash of the titans: Mapreduce vs. spark for large scale data analytics”. In: *Proceedings of the VLDB Endowment* 8.13 (2015), pp. 2110–2121.

[67] Mark Shifrin, Rami Atar, and Israel Cidon. “Optimal scheduling in the hybrid-cloud”. In: *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE. 2013, pp. 51–59.

[68] Jordi Soria-Comas and Josep Domingo-Ferrer. “Big data privacy: challenges to privacy principles and models”. In: *Data Science and Engineering* 1.1 (2016), pp. 21–28.

[69] Dan L. Starr et al. “A Map/Reduce Parallelized Framework for Rapidly Classifying Astrophysical Transients”. In: *Astronomical Data Analysis Software and Systems XIX*. Vol. 434. ASP Conference Series. 2010.

[70] Ashish Thusoo et al. “Hive: a warehousing solution over a map-reduce framework”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[71] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[72] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[73] Qi Wang and Xiaojun Huang. “PFT: A Performance-Fairness Scheduler on Hadoop YARN”. In: *Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on*. IEEE. 2016, pp. 76–80.

[74] *What is DC/OS?* URL: https://docs.mesosphere.com/1.11/overview/what-is-dcos/.

[75] Thomas Wirtz and Rong Ge. "Improving MapReduce Energy Efficiency for Computation Intensive Workloads". In: *Green Computing Conference and Workshops (IGCC), 2011 International*. 2011, pp. 1–8.

[76] Joing Xie et al. "Improving Mapreduce Performance Through Data Placement in Heterogeneous Hadoop Clusters". In: *IPDPS Workshops*. 2010, pp. 1–9.

[77] Yi Yao et al. "Haste: Hadoop yarn scheduling based on task-dependency and resource-demand". In: *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE. 2014, pp. 184–191.

[78] Shanhe Yi, Cheng Li, and Qun Li. "A survey of fog computing: concepts, applications and issues". In: *Proceedings of the 2015 Workshop on Mobile Big Data*. ACM. 2015, pp. 37–42.

[79] Chung-Tsz Yuan and Shenjian Chen. "Message Passing Interface (MPI)". In: (1996).

[80] Matei Zaharia et al. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[81] Matei Zaharia et al. “Improving MapReduce performance in heterogeneous environments”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. URL: http://dl.acm.org/citation.cfm?id=1855741.1855744.

[82] Eddy Z Zhang et al. “Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping”. In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM. 2010, pp. 115–126.

[83] Yang Zhang and Dan Liu. “Improving the efficiency of storing for small files in hdfs”. In: *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE. 2012, pp. 2239–2242.

[84] Ziliang Zong et al. “An Energy-Efficient Framework for Large-Scale Parallel Storage Systems”. In: *Parallel and Distributed Processing Symposium*. 2007, pp. 1–7.